, , 1–26 ()

Searching for a Global Search Algorithm

SABINE DICK sab@informatik.uni-bremen.de
Universität Bremen, FB3, Informatik, D-28344 Bremen, Germany

THOMAS SANTEN santen@first.gmd.de
German National Research Center for Information Technology (GMD FIRST), Rudower Chaussee 5, D-12489 Berlin, Germany

Received Oct 9, 1995

Editor:

Abstract. We report on a case study to assess the use of an advanced knowledge-based software design technique with programmers who have not participated in the technique's development. We use the KIDS approach to algorithm design to construct two global search algorithms that route baggage through a transportation net. Construction of the second algorithm involves extending the KIDS knowledge base. Experience with the case study leads us to integrate the approach with the spiral and prototyping models of software engineering, and to discuss ways to deal with incomplete design knowledge.

Keywords: Formal Methods, KIDS, Program Synthesis, Scheduling.

1. Introduction

Advanced techniques to support software construction will only be widely accepted by practitioners if they can be successfully used by software engineers who were not involved in their development and did not receive on-site training by their inventors. Experience has to be gained how knowledge-based methods can be integrated into the practical software engineering process.

We report on experience with the application of the approach to algorithm design underlying the *Kestrel Interactive Development System* (KIDS) [17] to the construction of control software for a simplified baggage transportation system at an airport. In this paper, we use the term *KIDS approach* to denote the concepts that have been implemented in the system KIDS. We did not use the implemented system KIDS in the case study, because we wanted to assess the approach rather than the system. In this way, we could exactly observe how concepts of the approach were used in the case study, and could modify the approach where necessary.

The KIDS system has been applied to a number of case studies at Kestrel Institute. In particular, it has been used in the design of a transportation scheduling algorithm with impressive performance [18], [19]. We wished to find out if we were able to use this method based on the available publications and produce satisfactory results with reasonable effort. A second goal of this work has been to study how a knowledge-based approach can be integrated into the overall software engineering

process. As a case study we chose a non-trivial abstraction of a practically relevant problem to make our experience transferable to realistic applications.

In the following, we elaborate on two issues: a process model we found useful to support application of the KIDS approach, and the merits and shortcomings of the approach we encountered when we explored alternative solutions to the transportation scheduling problem.

We have integrated the spiral and prototyping models of software engineering [2] with the KIDS approach. We developed the first formal specification and a prototype implementation in parallel. The prototype served to validate the specification and to improve understanding of the problem domain.

In the KIDS approach, global search algorithms are constructed by specializing global search theories that abstractly describe the shape of the search tree set up by the algorithm. For the case study, we wished to explore two alternative search ideas. While we found a theory suitable for the first one in the literature; the second one could not be realized with the documented design knowledge. This led us to develop a new global search theory that needs a slightly modified specialization procedure.

In Section 2, we introduce the baggage transportation problem. Section 3 provides a brief review of the global search theory and the KIDS approach. We present its integration into a process model in Section 4. The design of two transportation schedulers is described in Section 5. Optimizations are sketched in Section 6, where we also discuss the resulting algorithms. Related work is described in Section 7, and we summarize our experience with the approach in Section 8.

2. Baggage Transportation Scheduling

We wish to develop a controller for the baggage transportation system at an airport. Baggage is transported from check-in counters to gates, from gates to other gates, or from gates to baggage delivery points. The controller must schedule the bags through the network in such a way that each bag arrives at its destination in due time.

To simplify the problem, we do not consider on-line scheduling of a continuous flow of baggage fed into the system at the check-in counters, but schedule all baggage checked-in at a particular point in time.

2.1. Domain Model

We model the transportation net as a directed graph as shown in Figure 1. Check-in counters and baggage delivery counters, gates and switches are represented by nodes. We classify these in three kinds: *input nodes*, *transportation nodes* and *output nodes*. Check-in counters correspond to input nodes, switches to transportation nodes and baggage delivery counters to output nodes. Since gates serve to load and unload airplanes, we represent them by an input and an output node.

The edges of the graph model conveyor belts. The *capacity* of a belt is the maximum amount of baggage, the "total weight", that it can carry at a time. Its *duration* is the time it takes to carry baggage from the start to the end node.

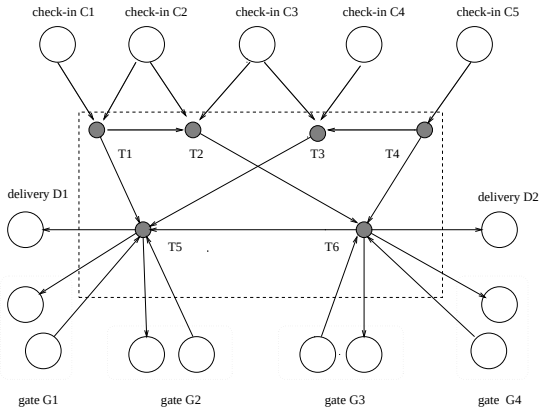

Figure 1. Transportation network

Bags are described by their *weight*, *source* and *destination* nodes, and their *due time*. Source and destination are input and output nodes, respectively. Weight and due time are positive natural numbers. Due times are specified relative to the beginning of the transportation process.

2.2. Problem Specification

We basically use the specification language of [17] which is sorted first order predicate logic. For notational convenience, we use simple polymorphism and higher-order functions in some places, and we also assume predefined sorts for standard data types such as the natural numbers, sets, and lists. Free variables in formulas are implicitly universally quantified and we leave out type information if it is clear from the context.

Our task is to assign, to each bag, a route through the network leading from its source to its destination node in due time. To keep things simple, we require an *acyclic* network without depots at the transportation nodes. Thus, the only way to resolve scheduling conflicts that arise if capacities of conveyor belts are exceeded is to delay baggage at source nodes. A *route* therefore is a pair of a delay and a *path* through the network. We model paths by sequences of vertices. A *plan* maps baggage to routes. We introduce abbreviations for these sorts, where $map(A, B)$ is the sort of all finite mappings from A to B.

$$\begin{aligned} path &= seq(vertex) \\ route &= nat \times path \\ plan &= map(baggage, route) \end{aligned}$$

Attempting to find a plan for a particular set of bags makes sense only if there exists a *feasible path* for each bag. This is a path p through the network g leading from the source node to the destination node of a bag b, i.e. the first vertex of p is the source node $source(b)$ of b and the last vertex of p is equal to the destination node $dest(b)$ of b. Furthermore, the capacities of all conveyor belts on the path must be at least as big as the weight of the bag b. This is expressed by the predicate $path\_can\_carry(g, p, b)$. A bag for which no feasible path exists cannot be scheduled through the transportation net. The notion of feasibility is formalized by the following equivalence.

$$\begin{aligned} &feasible\_path(g, b, p) \Longleftrightarrow \\ &\quad is\_path(g, p) \wedge source(b) = first(p) \\ &\quad \wedge dest(b) = last(p) \wedge path\_can\_carry(g, p, b) \end{aligned} \tag{1}$$

Feasibility considers only single paths of a plan. Two additional restrictions refer to the interaction of routes in a plan. First, a plan is acceptable only if the total weight of the bags on any belt at any time does not exceed the belt's capacity, which is expressed by the predicate $capacity\_bounded(g, p)$. Second, we are only interested in complete plans that schedule all baggage. Thus we require the domain of a plan to consist of the entire set of bags bs that must be scheduled, $bs = dom(p)$.

An ideal solution is a plan with feasible paths where all bags arrive at their destination nodes in time and it is not necessary to delay any bags at their source nodes. Although an ideal plan does not exist in general, we must find some plan as long as feasible paths exist for all bags to schedule. Thus, we have to find an optimality criterion to select a most appropriate plan with feasible paths.

For our problem, punctuality is most important, while it is desirable to minimize delays of bags at source nodes and thereby reduce storage space needed at the input counters. We consider punctuality and delays only in a qualitative way, and define a *cost* function based on the criteria if all bags are delivered in time and if baggage is delayed at input nodes.

cost	all bags in time	no delays
0	yes	yes
1	yes	no
2	no	yes
3	no	no

Imagine we have a suitcase b_1 at check-in counter C_1 and another one b_2 at gate G_4 in Figure 1. Both have weight 1. They are checked in for the same flight, leaving from gate G_2. Let the transportation time of each belt be one time unit and its capacity also be one unit. A valid transportation plan maps b_1 and b_2 to undelayed routes, where the paths through the net are described by sequences of nodes.

Function $transport\_plan(bs : set(baggage),\ g : graph)$
  where $acyclic(g) \wedge \forall b \in bs.\ \exists p.\ feasible\_path(g, b, p)$
  returns $(p : plan \mid min(cost, p, \{q \mid bs = dom(q) \wedge capacity\_bounded(g, q)$
    $\wedge \forall b \in dom(q).\ feasible\_path(g, b, snd(q(b)))\}))$

Figure 2. Problem Specification

$$p = \begin{pmatrix} b_1 & \mapsto & (0, \langle C_1, T_1, T_2, T_6, T_5, G_2 \rangle) \\ b_2 & \mapsto & (0, \langle G_4, T_6, T_5, G_2 \rangle) \end{pmatrix}$$

Now suppose suitcase b_1 shall be transported from C_4 to G_2. As we have to avoid exceeding the capacity of the belt leading to G_2, one possible solution is to delay b_1 by one time unit. This gives us the transportation plan:

$$p = \begin{pmatrix} b_1 & \mapsto & (1, \langle C_4, T_3, T_5, G_2 \rangle) \\ b_2 & \mapsto & (0, \langle G_4, T_6, T_5, G_2 \rangle) \end{pmatrix}$$

With the predicates defined so far, we can set up the problem specification as shown in Figure 2. We wish to synthesize a function called *transport_plan* with two input parameters, a set of baggage bs and a graph g. The **where** -clause describes the precondition of the function: we may assume that g is acyclic and that there is a feasible path in g for all bags in bs. The **returns** -clause describes the postcondition for the result of *transport_plan*. The predicate $min(f, x, s)$ is true if x is a member of the set s such that $f(x)$ is a minimum of the image of s under f. Thus, *transport_plan* has to select a plan p with minimal *cost* from the set of all plans q that are complete, do not put more load on belts than allowed, and where the path assigned to each bag b is feasible. This path is the second component $snd(q(b))$ of the route assigned to b under plan q.

3. Design of Global Search algorithms

The basic idea of algorithm design in the KIDS approach is to represent design knowledge in design theories. Such a theory is a logical characterization of problems that can be solved by an algorithmic paradigm like "divide and conquer" or "global search". Algorithm design consists of showing that a given problem is an instance of a design theory. In the following, we summarize how global search algorithms are developed in the KIDS approach. For a full account, we refer the reader to [15], [16], [17].

Note that the theory of global search algorithms has been developed at Kestrel for nearly a decade. Our work is based on information drawn from several publications which reflect different stages of the theory's development. The account of global search presented in this paper therefore is not a verbatim citation but we have made several minor changes.

3.1. Design Theory

The logical frameworks used in the KIDS approach are algebraic specifications or *theories*, and mappings between them. Theories consist of a *signature* and a list of *axioms* over the signature. The signature declares sorts and functions with their sorts. The axioms describe properties of the functions introduced in the signature. A *signature morphism* maps the sort and function symbols of one specification to expressions over the sorts and functions of another specification such that the sorts of the target expressions are consistent with the sorts of the source symbols. A signature morphism is called a *theory morphism* if the images of the axioms of the source theory are theorems in the target theory, i.e. they are logically entailed by the axioms of the target theory.

The class of problems we deal with is to synthesize a function that is correct with respect to a specification of its input/output behavior. A quadruple $\mathcal{P} = \langle D, R, I, O \rangle$ is called a *problem specification* if the following conditions hold. The sorts D and R describe the input domain and the output range of the function. The predicate $I : D \to Bool$ describes the admissible inputs and $O : D \times R \to Bool$ describes the input/output behavior. A function $f : D \to R$ is a solution to a problem $\mathcal{P}$ if

$$\forall x : D.\ I(x) \Longrightarrow O(x, f(x)) \tag{2}$$

The problem specification of the transportation problem is described by the following signature morphism:

$$\begin{aligned}
&f \mapsto transport_plan \\
&D \mapsto set(baggage) \times graph \\
&R \mapsto plan \\
&I \mapsto \lambda\langle bs, g\rangle.\ acyclic(g) \wedge \forall b \in bs.\ \exists p.\ feasible_path(g, b, p) \\
&O \mapsto \lambda\langle bs, g\rangle, p. \\
&\qquad min(cost, p, \{q \mid bs = dom(q) \wedge capacity_bounded(g, q) \\
&\qquad\qquad \wedge \forall b \in dom(q).\ feasible_path(g, b, snd(q(b)))\})
\end{aligned} \tag{3}$$

There is an obvious translation from Figure 2 to this morphism. The Cartesian product of the input parameters' sorts becomes the input domain D while R becomes the sort of the result in the **returns** -clause. The predicate of this clause is transformed into a function from $D \times R$ to $Bool$ by λ-abstraction over the pair of input parameters and the result.

The synthesis problem is solved if we find an expression for *transport_plan* such that the translation of formula (2) is a theorem under the theory of baggage and graphs.

A *design theory* extends a problem specification by additional functions. It states properties of these functions sufficient to provide a schematic algorithm that correctly solves the problem.

The basic idea of "global search" is to split *search spaces* containing candidate solutions into "smaller" ones until solutions are directly extractable. In this way, a global search algorithm constructs a search tree in which nodes represent search spaces and the children of a node are representations of its subspaces. Solutions may be extractable from each node of the tree, and the set of all solutions extractable from nodes of the tree is the set of all solutions to the search problem.

Let us consider an example. We wish to find a total mapping from the set $U = \{a, b\}$ to the set $V = \{c, d\}$ that fulfills some predicate O and is optimal with respect to some cost function c. Global search can be used to solve this problem as is illustrated in Figure 3. The idea is to generate *all* total mappings from U to V,

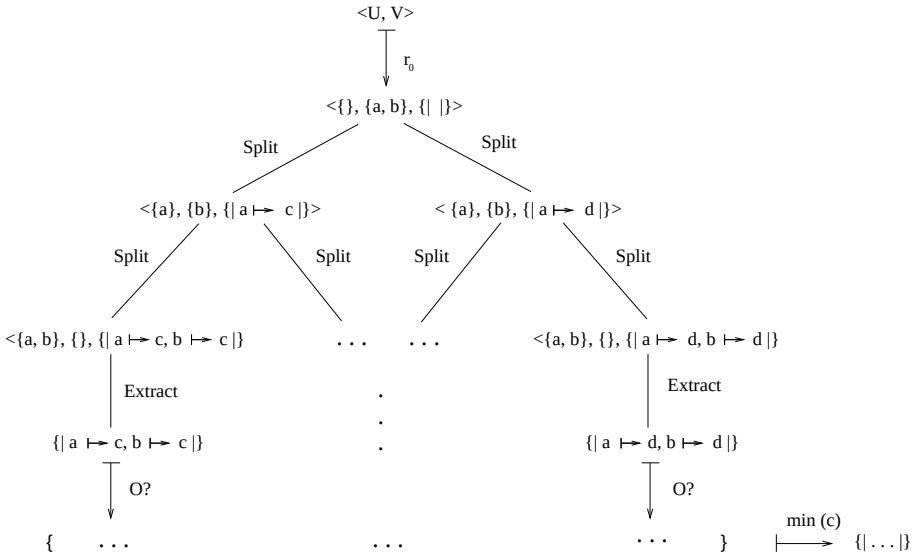

Figure 3. Global search for finite mappings

collect the ones that fulfill O and find one of these that has minimal cost.

Search spaces can be described by triples $\langle S, T, M \rangle$ where S and T partition U and M is a mapping from S to V. The set T contains the elements of U to which nothing has yet been assigned. In order to find all mappings from U to V, the initial search space which becomes the root of the search tree must be $\langle \{\}, U, \{\!\{\}\!\} \rangle$, where $\{\!\{\}\!\}$ is the empty mapping. The initial search space is constructed from the input by the function $\hat{r_0}$. Search is performed by splitting search spaces as long as T is not empty. The predicate Split relates a search space to its direct subspaces. They are obtained by picking arbitrary elements of T and V, and extending M by the corresponding maplet. The predicate Extract relates a search space descriptor to all solutions that are directly extractable from it. In our example, only the leaves of the search tree contain directly extractable solutions, namely the total mappings M. So far, we have only generated all mappings from U to V but not

checked if they fulfill O. This is done after extracting solutions, and finally we pick one of these that is minimal with respect to the cost function c. The search tree of Figure 3 shows the search idea for our example, and a global search theory for finite mappings which we present in Section 5.1 formalizes this idea.

For a concrete problem where O and c are known, there are many possibilities for optimization. It may be possible to prune branches of the search tree if they cannot contain feasible solutions, and the search can be terminated if we can decide on the optimality of a solution without inspecting all of them.

In the rest of this section, we describe an *abstract* global search theory that captures the essence of all (optimizing) global search algorithms. Every concrete search idea, such as the one discussed above, can then be formalized as a specialization of the abstract theory via a theory morphism. Two such specializations are used in Section 5.

The abstract global search theory is shown in Figure 4. It is an extension of

Sorts $D, R, \hat{R}, C$
Operations
$I : D \to Bool$
$O : D \times R \to Bool$
$\hat{I} : D \times \hat{R} \to Bool$
$\hat{r_0} : D \to \hat{R}$
$\mathsf{Satisfies} : R \times \hat{R} \to Bool$
$\mathsf{Split} : D \times \hat{R} \times \hat{R} \to Bool$
$\mathsf{Extract} : R \times \hat{R} \to Bool$
$c : R \to C$
$\leq : C \times C \to Bool$
Axioms
GS0. $I(x) \Longrightarrow \hat{I}(x, \hat{r_0}(x))$
GS1. $I(x) \wedge \hat{I}(x, \hat{r}) \wedge \mathsf{Split}(x, \hat{r}, \hat{s}) \Longrightarrow \hat{I}(x, \hat{s})$
GS2. $I(x) \wedge O(x, z) \Longrightarrow \mathsf{Satisfies}(z, \hat{r_0}(x))$
GS3. $I(x) \wedge \hat{I}(x, \hat{r}) \Longrightarrow$
$\mathsf{Satisfies}(z, \hat{r}) = (\exists \hat{s}.\ \mathsf{Split}^*(x, \hat{r}, \hat{s}) \wedge \mathsf{Extract}(z, \hat{s}))$
GSC. $\leq$ is total ordering on C

Figure 4. Abstract global search theory

the problem specification $\langle D, R, I, O \rangle$. The new sort $\hat{R}$ is the type of search space descriptors. The predicate $\hat{I}$ characterizes *legal* search space descriptors. In the example above, $\hat{I}$ contains the condition that S and T form a partition of U. For an input $x : D$, $\hat{r_0}(x)$ is the initial search space. Axiom GS0 ensures that the initial search space is legal.

The descendent relation on legal search spaces is given by Split. $\mathsf{Split}(x, \hat{r}, \hat{s})$ is true if $\hat{s}$ is a (direct) subspace of $\hat{r}$ for an input x. By axiom GS1, all children of

a legal subspace are again legal. The solutions that are obtainable from a single node $\hat{r}$ of the search tree are described by $\mathsf{Extract}(z, \hat{r})$.

By axiom GS3, $\mathsf{Satisfies}(z, \hat{r})$ describes the solutions z contained in a search space $\hat{r}$ that can be found with finite effort. We assume that x is a valid input,$I(x)$, and $\hat{r}$ is a legal descriptor for x, $\hat{I}(x, \hat{r})$. Then $\mathsf{Satisfies}(z, \hat{r})$ means that there exists a search space $\hat{s}$ from which we can extract z and that is connected to $\hat{r}$ by a *finite* path in the search tree. The latter condition is expressed by Split^* which is defined in the following way.

$$\begin{aligned}
\mathsf{Split}^*(x, \hat{r}, \hat{s}) &= (\exists k : nat.\ \mathsf{Split}^k(x, \hat{r}, \hat{s})) \\
\mathsf{Split}^0(x, \hat{r}, \hat{s}) &= (\hat{r} = \hat{s}) \\
\mathsf{Split}^{k+1}(x, \hat{r}, \hat{s}) &= (\exists \hat{t}.\ \mathsf{Split}(x, \hat{r}, \hat{t}) \wedge \mathsf{Split}^k(x, \hat{t}, \hat{s}))
\end{aligned}$$

Since we wish to find a *globally* optimal solution, GS2 requires that all feasible solutions z for a valid input x are contained in the initial search space $\hat{r_0}(x)$.

Finally, the sort C is the range of the cost function c. By axiom GSC, it is totally ordered with respect to the ordering relation $\leq$. Thus, it makes sense to define an optimal solution z as one with minimal cost $c(z)$.

Based on the theory of Figure 4, we can provide a schematic algorithm F that computes, for an input x with $I(x)$, a minimum cost solution z for which $O(x, z)$ holds. This algorithm is shown in Figure 5. The function fgs implements the

Function $F(x : D)$
  where $I(x)$
  returns $(z : R \mid min(c, z, \{z' \mid O(x, z')\}))$
  $= \mathit{fgs}(x, \hat{r_0}(x))$

Function $\mathit{fgs}(x : D, \hat{r} : \hat{R})$
  where $I(x) \wedge \hat{I}(x, \hat{r}) \wedge \Phi(x, \hat{r})$
  returns $(z : R | min(c, z, \{z' \mid \mathsf{Satisfies}(z', \hat{r}) \wedge O(x, z')\}))$
  $= \mathsf{some}(z : R \mid min(c, z, \{z' \mid \mathsf{Extract}(z, \hat{r}) \wedge O(x, z')\}$
  $\cup \{\mathit{fgs}(x, \hat{s}) \mid \mathsf{Split}(x, \hat{r}, \hat{s}) \wedge \Phi(x, \hat{s})\}))$

Figure 5. Global search algorithm schema

actual search and is called by F with the initial search space $\hat{r_0}(x)$. It returns some z with minimal cost that is either directly extracted from the parameter search space $\hat{r}$, i.e. it is an element of $\{z' \mid \mathsf{Extract}(z, \hat{r}) \wedge O(x, z')\}$, or that is obtained by splitting $\hat{r}$ and recursively applying fgs to its subspaces $\hat{s}$.

In addition to the elements of the abstract global search theory, this algorithm uses a *necessary filter* Φ to prune branches of the search tree. Necessary filters are defined by the implication

$$I(x) \wedge \hat{I}(x, \hat{r}) \wedge (\exists z : R.\ \mathsf{Satisfies}(z, \hat{r}) \wedge O(x, z)) \Longrightarrow \Phi(x, \hat{r}) \tag{4}$$

This means if $\Phi(x, \hat{r})$ does *not* hold then there are no feasible solutions in $\hat{r}$ and we need not search this space.

Note that the theory of Figure 4 does not guarantee termination of F. Infinite chains of split operations are possible as well as infinitely many subspaces of a search space. Furthermore, totality of F on valid inputs is not ensured because the global search theory does not require feasible solutions to exist for all valid inputs, i.e. it does not entail

$$\forall x : D.\ I(x) \Longrightarrow \exists z : R.\ O(x, z)$$

3.2. Algorithm Design

How can we find a global search algorithm for a given problem specification? We have to find a search space description $\hat{R}$ and operations $\hat{r_0}$, Satisfies, Split, and Extract such that the global search axioms are fulfilled.

In the KIDS approach this is done by referring to knowledge about search strategies on concrete data structures that is formalized in a library of general global search theories. We used Appendix A of [16] in our case study. Examples are theories to enumerate all sequences over a finite set and to enumerate all mappings between finite sets. A global search theory for a given problem is constructed by specializing a theory from the library. A problem theory $\mathcal{A} = \langle D_A, R_A, I_A, O_A \rangle$ *specializes* to a problem theory $\mathcal{B} = \langle D_B, R_B, I_B, O_B \rangle$ if

$$\begin{array}{l} R_B \subseteq R_A \\ \wedge\ \forall x : D_B.\ \exists y : D_A.\ \forall z : R_B.\ (I_B(x) \Longrightarrow I_A(y)) \\ \qquad \wedge\ (I_B(x) \wedge O_B(x, z) \Longrightarrow O_A(y, z)) \end{array} \tag{5}$$

Condition (5) basically says that every solution z for $\mathcal{B}$ is also a solution for $\mathcal{A}$. Thus, if we know how to construct solutions for $\mathcal{A}$ and we can easily decide if a solution for $\mathcal{A}$ also is a solution for $\mathcal{B}$, then we have found a way to construct solutions for $\mathcal{B}$. Figure 6 further illustrates this idea. We want to construct an algorithm f_B for this problem, i.e. we want to implement transition (a) at the bottom of Figure 6. Suppose that we know a solution f_A for $\mathcal{A}$ which implements transition (c). If $\mathcal{A}$ specializes to $\mathcal{B}$ then we know that we can find an input y of f_A for every input x of $\mathcal{B}$. Application of f_A gives us an $\mathcal{A}$-solution z (c). The final step (d) is to test if z fulfills O_B. In this way, the problem of finding an algorithm for $\mathcal{B}$ reduces to finding corresponding inputs in step (b) and deciding if O_B holds in step (d).

Verifying (5) and finding the mapping t from $\mathcal{B}$-inputs to $\mathcal{A}$-inputs can be done hand in hand if we construct a witness for the existential quantification over y while proving (5). This witness, in general, is a term depending on x which can be interpreted as a function from D_B to D_A.

Global search theories are extensions of problem theories. If we find a global search theory in our library whose problem theory specializes to the problem $\mathcal{B}$

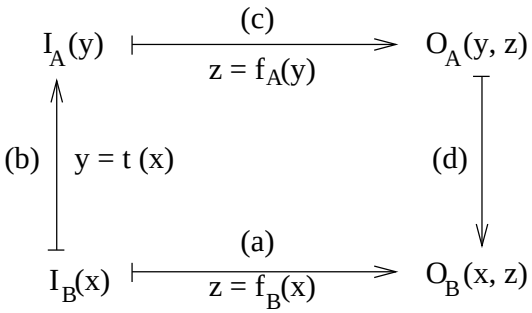

Figure 6. Specialization

at hand, we can extend $\mathcal{B}$ to a global search theory using $\hat{r_0}$, Satisfies, etc. of the library theory where we substitute the witness $t(x)$ for the input parameters. How this exactly works is shown in Section 5.

In general, the algorithm obtained by instantiating the schema of Figure 5 is very inefficient. But it has a high potential for optimization which can be exploited by deriving a choice of filters, by program transformation, and by data structure refinement. The optimizations we have applied in the case study are discussed in Section 6.

4. A Process Model

Our presentation of the application domain theory and problem specification in Section 2 only describes the final result of the specification effort. To develop the domain theory is one of the major tasks if not the most complex and time consuming one in the KIDS approach. Much of its complexity stems from two requirements we demand of the domain theory: it must not only make precise the informal — usually incomplete and sometimes inconsistent — ideas about the nature and context of the problem, but it must also be formulated so as to aid and not impede the subsequent design process. The KIDS approach does not provide direct help to construct domain theories. In contrast, it needs a formal domain theory to work on and can smoothly be applied only if the presentation of this theory has a suitable syntactic form.

As a consequence, it is very unlikely that a satisfactory domain theory can be built from scratch. This observation led us to integrate the KIDS approach with the prototyping and spiral models of software engineering [2]. One cycle of development, as sketched in Figure 7, has three phases. The first is concerned with establishing or enhancing the domain theory, the second produces code, and in the third phase code and theory are tested and validated.

We found it useful to build the first draft of the domain theory in parallel with a prototype. In this early phase, shaded gray in Figure 7, the domain theory is not

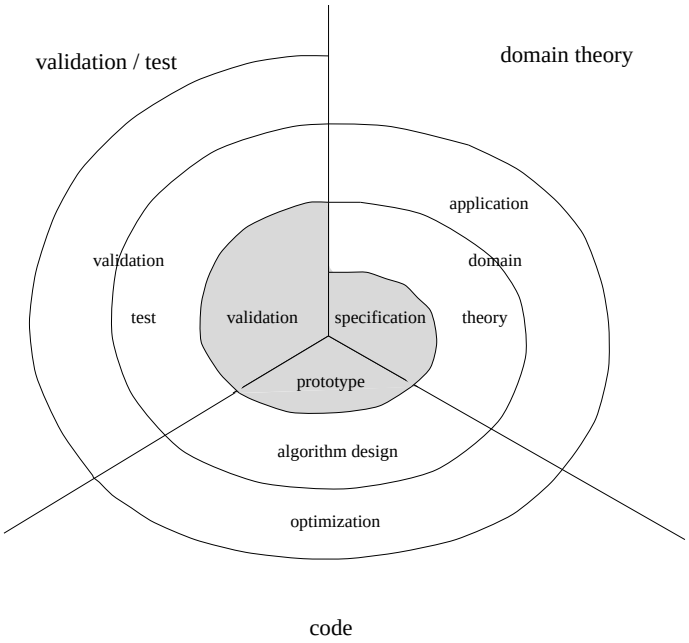

Figure 7. Process model

rich enough to apply algorithm design knowledge from design theories. Building a prototype enables us to get a deeper understanding of the problem and the essential properties of the application area. It helps us to build a complete domain theory and to avoid dead-end developments.

The way in which the domain model is expressed, the data structures used, and the properties stated, can have much influence on the ease with which algorithm design can be carried out. Thus, what seems to be one cycle of design in Figure 7, may in practice require several rounds of refining the domain theory until the formalized notions smoothly fit with the design theory we wish to use. This situation seems to be typical for *constructive* formal methods like program synthesis and transformation or interactive mechanical theorem proving. The more we wish to systematically construct solutions and provide tool support for this process, the more important becomes the syntactic presentation of our problems.

One example from the baggage transportation case study is the way we modeled delays in routes (cf. Section 2.2). In an early version of the domain theory, we described them by repetitions of the input nodes in paths, each occurrence of the input node denoting a delay of one time unit. This forced us to introduce predicates to characterize legal routes, and we could not use an acyclic graph model. When we decided to reformulate the theory and make delays explicit the theory became much more elegant and further design was much easier.

The process of theory refinement perpetuates as we derive filters and optimize code. The validation and test phases also serve us to validate the code with respect to properties that are not captured by the design knowledge put to our disposal in the KIDS approach. Examples for such properties are the adequacy of domain theories and the efficiency of the synthesized code.

5. Two Ways to Find Transportation Plans

Looking at the sort of transportation plans,

$$map(baggage, nat \times seq(vertex))$$

suggests two strategies to search for solutions to our scheduling problem.

1. *Domain Extension.* Start with the initially empty map and successively extend it by assigning possibly delayed feasible paths to baggage.
2. *Image Modification.* Start with the map that assigns their source nodes and no delay to all baggage; successively modify the assigned routes by extending paths or increasing delays.

Both strategies enumerate all feasible transportation plans. In the KIDS approach, search strategies are provided in a library of general global search theories. Algorithm design proceeds by specializing one of these to the problem at hand. The first condition in (5), $R_B \subseteq R_A$, suggests matching the output domains of the problem specification with the ones of the library theories to find candidates to specialize.

When we began algorithm design for the transportation problem, our initial idea was to use the image modification strategy, but there is no general global search theory documented in the KIDS library [16] that models image modification. Instead, we found a theory that describes domain extension. This motivated us to explore both approaches.

5.1. Domain Extension

Searching Appendix A of [16], we found a global search theory that specializes to the transportation problem. The theory *gs_finite_mappings* is a generalization of the search strategy to enumerate finite mappings shown in Figure 3 to arbitrary finite sets U and V. We present this theory by a theory morphism mapping the signature of the abstract global search theory of Figure 4 to concrete expressions.

$$\begin{aligned}
&F \mapsto gs_finite_mappings\\
&D \mapsto set(\alpha) \times set(\beta)\\
&R \mapsto map(\alpha, \beta)\\
&I \mapsto \lambda\langle U, V\rangle.\ |U| < \infty \wedge |V| < \infty\\
&O \mapsto \lambda\langle U, V\rangle, N.\ N \in Map(U, V)
\end{aligned}$$

$$
\begin{array}{ll}
\hat{R} & \mapsto set(\alpha) \times set(\alpha) \times map(\alpha, \beta) \\
\hat{I} & \mapsto \lambda\langle U, V\rangle, \langle S, T, M\rangle.\ S \uplus T = U \wedge M \in Map(S, V) \\
\mathsf{Satisfies} & \mapsto \lambda N, \langle S, T, M\rangle.\ \forall x \in S.\ N(x) = M(x) \\
\hat{r_0} & \mapsto \lambda\langle U, V\rangle.\ \langle \{\}, U, \{\!\|\}\rangle \\
\mathsf{Split} & \mapsto \lambda\langle U, V\rangle, \langle S, T, M\rangle, \langle S', T', M'\rangle. \\
 & \quad (\exists a, b.\ a = arb(T) \wedge b \in V \\
 & \quad \wedge \langle S', T', M'\rangle = \langle S + a, T - a, M \oplus \{\!|a \mapsto b|\!\}\rangle) \\
\mathsf{Extract} & \mapsto \lambda N, \langle S, T, M\rangle.\ T = \{\} \wedge N = M
\end{array}
$$

The first part of the morphism determines the problem theory. The input domain D is a pair of sets over arbitrary types α and β. The input condition I restricts the valid inputs $U : set(\alpha)$ and $V : set(\beta)$ to finite sets. The output range R consists of all finite mappings from α to β. Feasible outputs are characterized by the output condition O: the resulting mapping N must have domain U and range V. Formally the function Map is defined by

$$M \in Map(U, V) \Longleftrightarrow dom(M) = U \wedge \forall b \in dom(M).M(b) \in V \tag{6}$$

Note the difference between the sort $map(\alpha, \beta)$ and the term $Map(U, V)$: the latter denotes the set of all total mappings from U to V, and each of them has the sort $map(\alpha, \beta)$.

The rest of the morphism extends the problem theory to a global search theory. Search spaces $\hat{R}$ are denoted by triples where the first two components are sets over α and the last is a mapping from α to β. By $\hat{I}$, a particular search space is valid for inputs U and V only if S and T partition U, i.e. $S \uplus T = U$, and M is a total mapping from S to V.

When is a solution N contained in a (valid) search space $\langle S, T, M\rangle$, i.e. when does $\mathsf{Satisfies}(N, \langle S, T, M\rangle)$ hold? According to the idea of domain extension, M as it is determined so far must be compatible with N. This is true if the images of N and M are identical on S, which is the domain of M.

As discussed in Section 3.1, we get a direct subspace of $\langle S, T, M\rangle$ if we extend the domain of M by exactly one element. This means we determine an arbitrary element $a = arb(T)$ of T and some value b of V, and extend M by mapping a to b, $M \oplus \{\!|a \mapsto b|\!\}$. To obtain a valid search space, we move a from T into S. Finally, if T is empty we have found a total mapping from U to V which we can extract.

In order to use *gs_finite_mappings* to construct an algorithm for the transportation problem, we have to show that its problem specification specializes to (3). The first step (cf. the specialization condition (5)) is to ensure that the range sort of the transportation problem is a subsort of the range sort of *gs_finite_mappings*. Unifying the two sort expressions, we find substitutions for the type variables in *gs_finite_mappings*.

$$
\begin{array}{l}
\alpha \mapsto baggage \\
\beta \mapsto route
\end{array}
$$

With this substitution, we get an instance of *gs_finite_mappings* whose output domain is equal to the one of *transport_plan*. It remains to find expressions in g and bs for U and V so that we can prove the specialization condition relating the two theories.

$$\begin{array}{l}\forall\langle g, bs\rangle.\ \exists\langle U, V\rangle.\ \forall M.\\ (acyclic(g) \wedge \forall b \in bs.\ \exists p.\ feasible\_path(g, b, p) \Longrightarrow |U| < \infty \wedge |V| < \infty)\\ \wedge\ (acyclic(g)\\ \quad \wedge \forall b \in bs.\ \exists p.\ feasible\_path(g, b, p)\\ \quad \wedge \forall b \in dom(M).\ feasible\_path(g, b, snd(M(b)))\\ \quad \wedge bs = dom(M) \wedge capacity\_bounded(g, M)\\ \quad \Longrightarrow M \in Map(U, V))\end{array} \tag{7}$$

The last implication gives the clue to find expressions for U and V. If we compare the definition of Map with the premise of the implication, we find two matching conjuncts. First, $dom(M) = U$ matches $bs = dom(M)$ if we substitute bs for U. Second, $\forall b \in dom(M).\ M(b) \in V$ and $\forall b \in dom(M).\ feasible\_path(g, b, snd(M(b)))$ are similar. Regarding predicates as propositions of membership of their extensions

$$P(x) \Longleftrightarrow x \in \{y \mid P(y)\}$$

suggests to use the set of all feasible paths as a substitution for V. Unfortunately, $feasible\_path$ depends on a particular bag b, while V must contain candidate routes for all members of bs. Analyzing the situation might suggest to use

$$\{\langle n, p\rangle \mid \exists b \in bs.\ feasible\_path(g, b, p)\}$$

as a substitution for V because quantification is the only way to "hide" b in the expression for V. Only to consider paths that are feasible for *all* bags of bs is certainly not useful — this set typically is empty because there are bags with different sources or destinations. This analysis appears rather complex and there is a much simpler way to come to a substitution for V: expanding the definition (1) of $feasible\_path$, we find that only the conjunct $is\_path(g, p)$ does not depend on b. The remaining conditions for feasible paths are naturally exploited by developing a necessary filter, as we show at the end of the section.

Still, just to take the set of all paths in g is not restrictive enough to make V a *finite* set because it does not restrict the delay n of the routes in V. We therefore introduce an upper bound $md(g, bs)$ on delays, and define the set of all routes through g with a delay less then or equal to $md(g, bs)$ by

$$routes(g, bs) = \{\langle n, p\rangle \mid is\_path(g, p) \wedge n \leq md(g, bs)\}$$

With this definition, we finally come to use the sets

$$\begin{array}{l}U \mapsto bs\\ V \mapsto routes(g, bs)\end{array}$$

to specialize *gs_finite_mappings*.

Since there are feasible paths for all bags in bs (cf. the precondition of *transport_plan* in (3)), we can assign to $md(g, bs)$ the sum of the times needed to traverse a feasible path for each bag. Applying the substitution for α, β, U and V to *gs_finite_mappings* gives us a global search theory for *transport_plan*.

$$
\begin{array}{ll}
\hat{R} & \mapsto set(baggage) \times set(baggage) \times plan \\
\hat{I} & \mapsto \lambda\langle g, bs\rangle, \langle S, T, M\rangle.\ S \uplus T = bs \wedge M \in Map(bs, routes(g, bs)) \\
\mathsf{Satisfies} & \mapsto \lambda N, \langle S, T, M\rangle.\ \forall b \in S.\ N(b) = M(b) \\
\hat{r_0} & \mapsto \lambda\langle g, bs\rangle.\ \langle \{\}, bs, \{\!|\ |\!\}\rangle \\
\mathsf{Split} & \mapsto \lambda\langle g, bs\rangle, \langle S, T, M\rangle, \langle S', T', M'\rangle. \\
 & \quad (\exists b, r.\ b = arb(T) \\
 & \quad \wedge\ r \in routes(g, bs) \\
 & \quad \wedge\ \langle S', T', M'\rangle = \langle S + b, T - b, M \oplus \{\!|b \mapsto r|\!\}\rangle) \\
\mathsf{Extract} & \mapsto \lambda N, \langle S, T, M\rangle.\ T = \{\} \wedge N = M
\end{array}
$$

Note that "substituting" for U and V, which are bound variables, here means to apply the components of *gs_finite_mappings* to the pair of values for U and V and then abstracting over the input parameters of *transport_plan*.

Also note that the global search theory for *transport_plan* contains the problem (3) rather than the problem specification of *gs_finite_mappings*. In this way, the more restrictive pre- and postconditions are incorporated into the search algorithm when we instantiate the schema of Figure 5.

The resulting search strategy assigns complete routes to one bag after the other. Without further optimization, Split assigns arbitrary routes to bags, and only when a complete plan can be extracted is it tested whether the assigned routes are feasible. An obvious way to prevent infeasible assignments in the first place is to develop a necessary filter. Actualization of (4), and the fact that *capacity_bounded* is monotonic in domain extensions of M gives us

$$
\begin{array}{l}
\Phi \mapsto \lambda\langle g, bs\rangle, \langle S, T, M\rangle. \\
\quad (\forall b \in S.\ source(b) = fst(snd(M(b))) \wedge dest(b) = last(snd(M(b))) \\
\qquad \wedge\ path\_can\_carry(g, p, snd(M(b)))) \\
\quad \wedge\ capacity\_bounded(g, M)
\end{array}
$$

Here, we have found an easy way to incorporate the remaining conjuncts of $feasible\_path$ into the algorithm.

5.2. Image Modification

There is no global search theory documented in [14], [16], [17] that supports searching for maps by image modification. So we developed a new theory for this purpose. Note that there is no easy way to extend the "library" of global search theories of the implemented system KIDS because it is hard-coded into the implementation.

Hence, we could not have used the system to work with the theory on image modification decribed in this section.

Abstracting from the concrete scheduling problem, the image modification strategy can be sketched as follows: The images of a given map (the initial schedule) are increased along the various degrees of freedom that are given by the range type of the map. A suitable successor relation on the elements of the range type can be used to describe the "direction" in which to increase the images of the map. This idea is formalized in *gs_parallel_mappings*.

$$
\begin{array}{ll}
F & \mapsto gs\_parallel\_mappings \\
D & \mapsto map(\alpha,\beta) \times set(\beta \times \beta) \\
R & \mapsto map(\alpha,\beta) \\
I & \mapsto \lambda\langle M,S\rangle.\ |dom(M)| < \infty \wedge \forall x.\ \neg(x\ S\ x) \\
 & \quad \wedge (\forall x,y.\ x\ S\ y \Longrightarrow \neg(\exists z.\ x\ S\ z \wedge z\ S\ y)) \\
O & \mapsto \lambda\langle M,S\rangle, N.\ dom(M) = dom(N) \wedge \forall x \in dom(M).\ M(x)\ S^*\ N(x) \\
\hat{R} & \mapsto map(\alpha,\beta) \times set(\beta \times \beta) \\
\hat{I} & \mapsto \lambda\langle M,S\rangle, \langle M',S'\rangle.\ S{=}S' \wedge dom(M) = dom(M') \\
 & \quad \wedge \forall x \in dom(M).\ M(x)\ S^*\ M'(x) \\
\mathsf{Satisfies} & \mapsto \lambda N, \langle M,S\rangle.\ (\forall x \in dom(M).\ M(x)\ S^*\ N(x)) \\
\hat{r_0} & \mapsto \lambda\langle M,S\rangle.\ \langle M,S\rangle \\
\mathsf{Split} & \mapsto \lambda\langle M,S\rangle, \langle M',S'\rangle, \langle M'',S''\rangle. \\
 & \quad (\exists x,y.\ x = arb(dom(M')) \wedge M'(x)\ S\ y \\
 & \quad \wedge \langle M'',S''\rangle = \langle M' \oplus \{\!|x \mapsto y|\!\}, S\rangle) \\
\mathsf{Extract} & \mapsto \lambda N, \langle M,S\rangle.N = M
\end{array}
$$

The inputs are a map M and a binary relation S on the range of M. We model the relation by a set of pairs. The input condition I requires that the domain of M is finite and that S is irreflexive and non-dense. We use $x\ S\ y$ as a notation for $\langle x,y\rangle \in S$ to increase readability. Non-density means that there is no z "between" any two values x and y with $x\ S\ y$. This is necessary to ensure that all solutions can be found by finitely many applications of Split. Since we just want to modify the image of input M, a solution N is a mapping with the same domain as M. Furthermore, each image $N(x)$ is reachable from the corresponding input image $M(x)$ by S, i.e. the pair $\langle M(x), N(x)\rangle$ lies in the reflexive and transitive closure S^* of S.

Search spaces of $\hat{R}$ are pairs of maps and relations. The invariant $\hat{I}$ and the initial search space $\hat{r}$ — which is the identity function — tell us that the relation is always the input relation S. We must include S in search spaces because we have to refer to it in Satisfies: a solution N is contained in a search space with mapping M if the images of N are reachable from M by S^*. The remainder of $\hat{I}$ is the output condition $O(M,M')$ for the maps in the search spaces. This shows that — unlike with *gs_finite_mappings* — we can extract a solution from every search space, namely its first component.

To split a search space $\langle M', S' \rangle$, we choose an arbitrary element x of the domain of M' and a direct successor y of $M'(x)$ under S. The new map M'' is constructed by overriding the image of x under M' by y.

The theory *gs_parallel_mappings* is a very abstract formalization of the idea searching for a mapping by image modification. There are only few restrictions on the successor relation S which determines the way search is actually performed. We could only put more restrictions on S if we made assumptions on the particular structure of the range sort β. But the resulting theory would be more specialized than is necessary to capture the search idea. This would restrict the range of problems to which it could be applied and is therefore undesirable.

5.2.1. Data Type Driven Specialization

Since we now have a theory that captures our search idea, we proceed by specializing the theory to the transportation problem as we did in Section 5.1. The instance of the specialization condition (5) reveals a drawback of the generality of *gs_parallel_mappings*.

$$
\begin{array}{l}
map(\alpha, \beta) \subseteq plan \\
\wedge \\
\forall \langle g, bs \rangle.\ \exists \langle M, S \rangle.\ \forall N.\ acyclic(g) \wedge \forall b \in bs.\ \exists p.\ feasible\_path(g, b, p) \\
\quad \Longrightarrow |dom(M)| < \infty \wedge \forall x.\ \neg(x\ S\ x) \\
\qquad\quad \wedge\ (\forall x, y.\ x\ S\ y \Longrightarrow \neg(\exists z.\ x\ S\ z \wedge z\ S\ y)) \\
\wedge \\
\quad (acyclic(g) \wedge \forall b \in bs.\ \exists p.\ feasible\_path(g, b, p) \\
\quad \wedge \forall b \in dom(N).\ feasible\_path(g, b, snd(N(b))) \\
\quad \wedge bs = dom(N) \wedge capacity\_bounded(g, N) \\
\quad \Longrightarrow dom(M) = dom(N) \wedge (\forall x \in dom(M).\ M(x)\ S^*\ N(x)))
\end{array}
\tag{8}
$$

This condition does not help much in systematically finding a substitution for M and S. Since *gs_parallel_mappings* does not make assumptions on the structure of β, comparing the syntactic structure of the input/output conditions of the global search theory and the transportation problem does not provide candidates for S and M.

At this point of the development, we could appeal to human intuition, invent substitutions for S and M, and verify that they are witnesses for (8). But this would contradict the general KIDS approach of systematically constructing unknowns wherever possible. Instead, we propose to specialize *gs_parallel_mappings* in two steps. The first step determines a suitable successor relation S while the second step finds a substitution for M.

To determine S, we first analyze the condition $map(\alpha, \beta) \subseteq plan$ to find substitutions for the type variables α and β. Unification of the two sorts yields

$$
\begin{array}{l}
\alpha \mapsto baggage \\
\beta \mapsto nat \times seq(vertex)
\end{array}
$$

Now we can analyze the range type $nat \times seq(vertex)$ to find a successor relation on its elements based the basic types it is composed of. We know the usual successor function on natural numbers, and a canonical way to extend sequences is to append an element. In analogy to lexicographical orderings on pairs, we construct a successor relation by extending either element of a pair. Thus, we define S by

$$\begin{array}{l} \langle n, p\rangle \; S \; \langle m, q\rangle \Longleftrightarrow \\ \quad (n + 1 = m \wedge p = q) \vee (n = m \wedge (\exists v.\; p +\!\!+ \langle v\rangle = q)) \end{array} \tag{9}$$

This definition fulfills the conditions on S in (8), namely irreflexivity and non-denseness. For the reflexive and transitive closure of S we get

$$\langle n, p\rangle \; S^* \; \langle m, q\rangle \Longleftrightarrow n \leq m \vee \exists y.\; p +\!\!+ y = q \tag{10}$$

With this definition for S^* it remains to show

$$\begin{array}{l} \forall \langle g, bs\rangle.\; \exists M.\; \forall N. \\ (acyclic(g) \wedge \forall b \in bs.\; \exists p.\; feasible\_path(g, b, p) \Longrightarrow |dom(M)| < \infty) \\ \wedge \\ (acyclic(g) \wedge \forall b \in bs.\; \exists p.\; feasible\_path(g, b, p) \\ \wedge \forall b \in dom(N).\; feasible\_path(g, b, snd(N(b))) \\ \wedge\; bs = dom(N) \\ \wedge\; capacity\_bounded(g, N) \\ \Longrightarrow dom(M) = dom(N) \\ \quad \wedge \forall x \in dom(M).\; fst(M(x)) \leq fst(N(x)) \\ \qquad\qquad \vee \exists q.\; snd(M(x)) +\!\!+ q = snd(N(x))) \end{array} \tag{11}$$

As in Section 5.1, we can now easily determine a substitution for M. It assigns to each b in bs the non-delayed path only consisting of the source node of b.

$$M \mapsto \{\!|b \mapsto \langle 0, \langle source(b)\rangle\rangle \mid b \in bs|\!\}$$

With this information, we can finally set up the image modification theory for the transportation problem like we did in Section 5.1.

6. Optimization and Results

The algorithms resulting from the instantiation of the schema in Figure 5 are well structured but very inefficient. Optimization of these algorithms is mandatory. Three classes of optimizations suggest themselves: filter development, program transformations, and refinement of data structures. The implemented system KIDS supports program transformations such as finite differencing and case distinction. It also supports developing necessary filters. In [16], various notions of filters for global search algorithms are formalized, and it is suggested to implement search heuristics by using priority queues to store search space descriptors.

Performing program transformations and data structure refinements on a formal basis is very costly and nearly impossible without machine support. We therefore decided to use the above optimization techniques as guidelines for our implementation but to build the implementation in an "ad-hoc" way without using formal techniques. We chose the functional programming language ML for the final implementation because functional programming offers an easy way to express instances of the program schema of Figure 5 in a programming language.

The strongest necessary filter which we have found for the algorithm of Section 5.1 is based on *capacity_bounded*: only partial plans which do not exceed the transportation capacity of any belt can be extended to complete, feasible plans. A stronger filter deciding if delaying a path can lead to a solution would be desirable, but without further assumptions on the transportation net such a filter is not obvious.

When we split a search space the test on *capacity_bounded* for the extended plan need only inspect the belts that are contained in the newly added route. This optimization can be regarded as an application of the "case distinction" transformation.

For the algorithm of Section 5.2 the situation is more complicated: the predicate *capacity_bounded* is not monotonic in increasing delays and can therefore not serve as a necessary filter. According to the successor relation S defined in (9), Split increases delays by one to generate direct subspaces. If *capacity_bounded* does not hold for the generated plan it may nevertheless hold for plans where the delay is increased by more than one unit. Consequently, we optimize Split and generate plans with minimally increased delays such that *capacity_bounded* holds.

Both algorithms need information that can be computed from the input once and for all, e.g. the feasible paths for all bags depend only on the input transportation net and not on search spaces. This information is precomputed before starting the search, which is an example application of finite differencing.

As suggested in [16], we use a priority queue to store search spaces. Crucial for efficiency is the choice of an ordering for the queue. The cost of the partial plans obviously has highest priority, but the classes of plans with equal costs are large and a finer ordering is needed. We have implemented the heuristic of searching "more complete" spaces first. For the domain extension algorithm, plans with larger domains have priority, while for the image modification algorithm, we prefer plans with longer paths and smaller total delays.

Finally, it turns out that the implementation of basic data structures like mappings and sets also has a great effect on the total performance. It is tempting to closely stay with the abstract data structures used in algorithm design and use a library of implementations for maps and sets. Implementing sets and tuples by lists, and using so-called *splay dicts* of the Standard ML library to implement maps, however, significantly increased performance.

In spite of these optimizations, performance of both algorithms is still poor, and only small examples can be treated in reasonable time. The major obstacle to better performance is that resolving conflicts by delays results in an extremely high branching factor of the search tree.

Test runs show that the image modification algorithm is in general faster than the domain extension algorithm. The difference in performance increases with the branching degree of the transportation net. We believe this justifies the extra effort needed to develop the global search theory of Section 5.2.

The domain and algorithm theories each have a size of a few hundred lines of formal text including specifications and theorems, based on a library of data structure specifications of approximately 500 lines. The resulting ML code has about 1000 lines of code. Approximately half of the ML code implements the data structures for the transportation net, while the rest implements the actual scheduling algorithms. The code is well-structured and highly reusable. Both algorithms share most of the code which facilitates exploring alternatives.

The case study required an effort of approximately 9 person months. We spent about one third of that time to learn the KIDS approach. Approximately 75% of the remaining time was devoted to building the domain theory.

7. Related Work

Approaches to algorithm design and program synthesis roughly fall in two categories. The ones advocating a calculational style of program development like the Dijkstra/Gries method [6], the refinement calculus [12] or Dershowitz's approach [4] are tuned towards application by hand. Others like deductive program synthesis [1], [11], program construction based on type-theory [3] and the approaches introduced in [10] focus on machine-supportable techniques. However, most of them, mechanized or not, are oriented on the syntactical structure of logical formulas or programming language constructs. They provide rules, e.g., to construct loops, or they describe how to synthesize programs, e.g., from specifications with conjunctive postconditions.

The KIDS approach differs from these in that it provides design steps reflecting *significant* design decisions. Algorithm theories abstract from implementation details for a particular programming language and characterize classes of algorithms (not programs) by logical theories. To achieve the effect of a design step like global search for a particular problem, many rule applications would be needed in other calculi, and — more important — these steps would have to re-invent the principles of search algorithms. The proof obligations arising would incorporate correctness conditions of global search — intertwined with conditions on the particular application domain and the specific code produced. Abstracting from these details, KIDS separates algorithm design from optimization concerns, and makes design knowledge amenable to re-use not on the level of code but on the conceptual level.

There are several attempts to improve confidence in the correctness of the synthesized algorithms by mechanically verifying the underlying theory. Kreitz [9] has formalized global search in the Nuprl type theory [3]. He specifically addresses the problem of termination and prevents infinite branchings of the search tree by using only finite sets in his formalization. He introduces *wf-filters* to prune infinite branches and proposes to provide a collection of wf-filters for each theory.

The value of mechanical verification of design theories and program transformations is much reduced if the relation between tools supporting the design process and the underlying theories is not clear. The approach to build transformation systems presented in [8] uses a generic theorem prover as the kernel of such a system. The prover is used to verify design theories and transformations as well as to support their application. This guarantees that the applied transformations are exactly the ones that have been verified. The approach is illustrated by proving and applying a global search theory on the basis of the generic theorem prover Isabelle [13].

It seems to be unlikely to find "practically complete" knowledge bases for software construction systems. Such systems should be designed to ease routine extension of their knowledge bases so they can be adapted to specific application domains and grow with the users' experience. In [7], a generic system architecture based on the notion of *strategies* is proposed. Strategy modules have a clearly defined interface to the system kernel, so new ones can be integrated into the system in a routine way. The system Specware [20] under development at Kestrel also seems to allow for a modularized and easily extendible knowledge base.

Our case study relates to the research on design of transportation schedulers at Kestrel [18], [19]. They study schedulers that assign trips to resources like planes, ships, and trucks to meet movement requirements. In this setting, trips fully occupy resources for an interval of time, i.e. the load of a resource cannot be extended during a trip. Furthermore, a trip changes the availability of a resource: the destination of one trip becomes the source of the next one. In baggage transportation, however, load of resources can continually change as baggage flows through the net, but source and destination points of a resource remain fixed in time.

Another difference lies in the focus of our work. For several years, a highly specialized theory on transportation scheduling has been developed at Kestrel with the aim to produce extremely efficient schedulers. Recently, this has even led to a refinement of the abstract global search theory [19]. The purpose of our case study, in contrast, has been to study in how far the KIDS approach as documented in the literature can support programmers who have no particular experience with the approach, to design algorithms for a non-trivial problem.

8. Discussion

We focus the discussion of our experience with KIDS on three questions: why use a formal approach instead of ordinary programming; what are the peculiarities introduced by formality; and what are the distinct advantages and disadvantages of KIDS?

8.1. Why Use a Formal Approach at all?

It is hard to speculate about the results an experienced programmer might have produced who started with knowledge about the problem domain similar to ours

when we began the case study. But we can point out where formality helped us in the case study. In the first circle of the process model of Figure 7, we implemented a prototype in an "ad-hoc" way to gain experience with the problem domain and come up with a first formal specification. Given our poor understanding of the problem at that time, the prototype revealed many aspects we had not been aware of. Formulating a specification afterwards and trying to identify these aspects of the problem in the specification forced us to search for a suitable level of abstraction to reason about the problem domain.

Having a *formal* specification and proving properties about it in the subsequent development revealed problems like overlapping search spaces and non-termination that we might have missed by just testing an implementation. Furthermore, testing was possible only after considerable optimization because the execution times — and traces — of non-optimized algorithms for the transportation problem were overwhelmingly large.

Concerning errors in an optimized program, the question arises whether a bug stems from the design or from the optimization. We believe that, although an experienced programmer with the right intuition might have been able to solve these problems with the program as the only "formal" document, formality helps in finding errors early and identifying their sources.

A formal specification not only provides an unambiguous, abstract documentation and the possibility to prove correctness of code relative to the specification. With the KIDS approach, design *decisions* — to construct a global search algorithm, to use image modification, what optimizations to apply — and their justifications are precisely documented as well.

8.2. Peculiarities Introduced by Formality

Some of the following observations may apply to formal methods in general, while arguably in a stronger sense to the KIDS approach because it is constructive and poses stronger requirements on the documents it deals with than mere "formal notations" or verification-based approaches.

Due to their preciseness, the logical theories on which the approach is based provide good reference points for software engineers who wish to learn and use them. Still, the theory of global search algorithms is inherently complex and it takes considerable effort to get a working knowledge of its application that enables one to map a particular problem to its formal representation.

The approach requires the existence of a formal problem specification. It does not directly address the first phase of the development, before a sufficiently complete application domain theory is available, which may be the most complex part of the process. We found prototyping useful to understand the problem domain, but more elaborate techniques to guide theory development remain to be established.

Moreover, the design theory one wants to apply later also influences theory development: the domain theory must supply the "right" notions and must be syntactically structured in a way that matches the design theory. For instance, defining

plans as mappings from baggage to routes helps to apply the finite mappings design theory because of the similar type structure.

If the syntactic difference between the domain and design theories is too large, the constructive approach may be difficult to follow even if the domain theory semantically captures all necessary requirements. Several component predicates and properties of *feasible_path* defined in our domain theory serve to establish a terminology to adequately formulate instances of the global search axioms. These predicates strongly depend on the way Split is axiomatized. Thus a thorough understanding of the design theory is necessary to focus theory development.

Given a domain theory, the steps in designing a global search algorithm: specializing a theory, deriving filters, and applying optimizing program transformations, provide a clear separation of concerns. Specialization determines the basic structure of the search, necessary filters exploit properties of the application domain, and only the final program transformations and data type refinements eliminate redundancies in the code and "fuse" filters with the basic search structure to gain efficiency.

Each of these tasks corresponds to one cycle in the process model that we introduced in Section 4. Thus the model helps programmers to focus activities on a particular task and to avoid introducing certain design ideas at the "wrong" time into the development. In early attempts to design the algorithm of Section 5.2, we tried to introduce optimizations too early — trying to generate delayed routes only if necessary — which made our design much too complex.

8.3. KIDS Specific Issues

Let us review our decision not to use the implemented system KIDS but to try and apply the underlying "approach" manually. Initially, this decision was motivated by the steep learning curve we expected a complex system like KIDS to have. We also wanted to make sure that we would be able to track down difficulties with the case study to their proper sources: peculiarities of the case study or problems inherent in the KIDS approach. If we had relied on the system then we could not have ruled out problems stemming from the implementation being a research prototype, or from our lacking experience using the system. To do so, we would have had to reconstruct the workings of the system, basically doing what we actually did when applying the approach manually.

In retrospect, we believe this decision is justified. Although we lost the possibility to actually apply optimizing program transformations, for the main task, namely to work with the global search design theory, our decision proved advantageous. The entire development of the image modification algorithm in Section 5.2 would not have been possible if we had confined ourselves to the working system. The system does not have an open design, and introducing a new global search theory would have been impossible for us to do. Furthermore, we would have needed to modify the specialization procedure to use the new theory, which again is a major programming task. Both, the approach and the system lack support for

constructing new algorithm theories and incorporate them into the working system. These non-trivial tasks deserve support if the approach shall be applied routinely, because for routine applications tool support is indispensable.

Our experience with the image modification algorithm shows that it is advisable to stick to the approach even if no design theory supporting a particular design idea is available. In this situation, it pays to develop a new design theory that describes the desired search strategy in an abstract way. In [5], we decided to construct the problem specific algorithm theory of Section 5.2 in one step and to manually verify it against the abstract global search theory. This decision was mainly due to lack of experience and increased the complexity of the task considerably. Moreover, it led to a less efficient algorithm.

A problem of a more technical nature is that termination of the constructed algorithms is not addressed by the global search theory we have used. This lead us to the somewhat unnatural introduction of the upper bound $md(g, bs)$ on delays (cf. Section 5). Termination of global search algorithms can be spoiled in two ways. There may be branches of the search tree with infinite length, or there may be nodes with infinitely many children. In [17], a well-founded ordering is introduced into the abstract global search theory to prevent infinite chains of Split-operations. There are reasons not to require termination of all global search theories — termination may be addressed only when a library theory is specialized — but we would appreciate a systematic way that relieves programmers of dealing with termination on-the-fly.

As many formal techniques, KIDS cannot deal explicitly with non-functional requirements such as efficiency and maintainability, but the formal theory of filters in [16] guides the search for possible optimizations. Since the approach is constructive, domain theories and code are well-structured and well-documented. This enhances requirements traceability and maintainability. Furthermore, with a domain theory at hand, the KIDS approach is well suited to construct prototypes in little time, and to explore alternative designs.

Finally, KIDS is one of the few approaches combining formality with the representation of software construction steps of considerable complexity. A library of algorithm theories makes standard design knowledge explicit and formally accessible. Application of design theories takes place at a much higher level of abstraction than with the language oriented rules many other formal program design calculi supply. We believe that, in correspondence with conceptual developments like design patterns and software architectures in software engineering in general, formal methods have to adapt larger patterns of reasoning and build a theory of software construction that allows reasoning at abstract, problem oriented levels. KIDS shows a step in this direction.

Acknowledgments

We would like to thank David Basin, Maritta Heisel and Burkhart Wolff for fruitfull discussions. David, Maritta, Klaus Didrich and Martin Simons provided comments

on drafts of this paper. Thanks also to the reviewers' whose comments helped to improve the presentation.

References

1. W. Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, 14:243–261, 1980.
2. B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
3. R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.
4. N. Dershowitz. *The Evolution of Programs.* Birkhäuser, 1983.
5. S. Dick. Eine Fallstudie zur Entwicklung korrekter Software: Steuerung einer Gepäckförderanlage. Master's thesis, Dept. of Computer Science, Technical University of Berlin, 1994.
6. D. Gries. *The Science of Programming.* Springer-Verlag, 1981.
7. M. Heisel, T. Santen, and D. Zimmermann. Tool support for formal software development: A generic architecture. In W. Schäfer and P. Botella, editors, *Software Engineering – ESEC '95*, LNCS 989, pages 272–293. Springer Verlag, 1995.
8. Kolyang, T. Santen, and B. Wolff. Correct and user-friendly implementation of transformation systems. In *FME '96 — Industrial Benefits and Advances in Formal Methods*, LNCS. Springer Verlag, 1996.
9. C. Kreitz. *Meta-Synthesis. Deriving Programs that Develop Programs.* Technische Hochschule Darmstadt, 1993.
10. M. Lowry and R. D. McCartney, editors. *Automating Software Design.* AAAI Press, Menlo Park, 1991.
11. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2:90–121, 1980.
12. C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.
13. L. C. Paulson. *Isabelle – A Generic Theorem Prover.* LNCS 828. Springer Verlag, 1994.
14. Thomas T. Pressburger, L. Gilham, and D. R. Smith. *Kestrel Interactive Development System, Version 1.0.* Kestrel Institute, 1991.
15. D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. In J. van de Snepscheut, editor, *Proc. International Conference on Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 379–398. Springer Verlag, 1989.
16. D. R. Smith. Structure and design of global search algorithms. Technical Report Kes.U.87.12, Kestrel Institute, 1987.
17. D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.
18. D. R. Smith and E. A. Parra. Transformational approach to transportation scheduling. In *Proceedings of the Eighth Knowledge-Based Software Engineering Conference*, Chicago, September 1993.
19. D. R. Smith, E. A. Parra, and S. J. Westfold. Synthesis of high-performance transportation schedulers. Technical Report KES.U.95.6, Kestrel Institute, 1995.
20. J. V. Srinivas and R. Jüllig. Specware: formal support for composing software. In *Proceedings of the Thrid Conference on Mathematics of Program Construction*, 1995.