AFRL-IF-RS-TR-1998-200
Final Technical Report
October 1998

# PLANWARE -- DOMAIN-SPECIFIC SYNTHESIS OF HIGH-PERFORMANCE SCHEDULERS

**Kestrel Institute**

**Lee Blaine, Limei Gilham, Jumbo Liu, Douglas R. Smith, and Stephen Westfold**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-200 has been reviewed and is approved for publication.

APPROVED: Robert J Paragi

ROBERT J. PARAGI
Project Engineer

FOR THE DIRECTOR:

NORTHRUP FOWLER III, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | *Form Approved*<br>*OMB No. 0704-0188* |

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| | | |
|---|---|---|
| **1. AGENCY USE ONLY** *(Leave blank)* | **2. REPORT DATE**<br>October 1998 | **3. REPORT TYPE AND DATES COVERED**<br>Final Jan 95 - Jun 97 |

**4. TITLE AND SUBTITLE**

PLANWARE -- DOMAIN-SPECIFIC SYNTHESIS OF HIGH-PERFORMANCE SCHEDULERS

**5. FUNDING NUMBERS**

C - F30602-95-C-0036
PE - 62702F
PR - 5581
TA - 27
WU - 85

**6. AUTHOR(S)**

Lee Blaine, Limei Gilham, Jumbo Liu, Douglas R. Smith, and Stephen Westfold

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Kestrel Institute
3260 Hillview Ave
Palo Alto CA 94304

**8. PERFORMING ORGANIZATION REPORT NUMBER**

N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

AFRL/IFTD
525 Brooks Road
Rome NY 13441-4505

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

AFRL-IF-RS-TR-1998-200

**11. SUPPLEMENTARY NOTES**

AFRL Project Engineer: Robert J. Paragi/IFTD/(315) 330-3547

**12a. DISTRIBUTION AVAILABILITY STATEMENT**

Approved for public release; distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

The technology aggregated in Planware represents part of a significant long-term investment in knowledge-based software engineering formulated and guided by AFRL/IFTD. It includes search theories from artificial and methodology for rigorous statement and formally-verified manipulation of specifications from the formal methods area with the goal of achieving a software engineering paradigm that supports the economic, life-long evolution of complex systems. Planware is a domain-specific generator of high-performance scheduling software, currently being developed at Kestrel Institute. Architecturally, Planware is an extension of the Specware system with domain-independent and domain-dependent parts. The domain-independent part includes a general algorithm design facility (including mechanisms to synthesize global-search and constraint propagation algorithms), as well as support for theorem-proving and witness finding. The domain-dependent part includes scheduling domain knowledge and architecture representations, and other domain-specific refinement knowledge that relates the scheduling domain to general algorithm design and data type refinement. Using Planware, the user interactively specifies a problem and then the system automatically generates a formal specification and refines it. Various transportation schedulers have been generated, and with satisfactory performance.

**14. SUBJECT TERMS**

Program synthesis, formal specification, planning, scheduling, domain-specific software synthesis, domain modeling

**15. NUMBER OF PAGES**

20

**16. PRICE CODE**

| | | | |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT** | **18. SECURITY CLASSIFICATION OF THIS PAGE** | **19. SECURITY CLASSIFICATION OF ABSTRACT** | **20. LIMITATION OF ABSTRACT** |
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

# PLANWARE – DOMAIN-SPECIFIC SYNTHESIS OF HIGH-PERFORMANCE SCHEDULERS

*Abstract*
Planware is a domain-specific generator of high-performance scheduling software, currently being developed at Kestrel Institute. Architecturally, Planware is an extension of the Specware system with domain-independent and domain-dependent parts. The domain-independent part includes a general algorithm design facility (including mechanisms to synthesize global-search and constraint propagation algorithms), as well as support for theorem-proving and witness finding. The domain-dependent part includes scheduling domain knowledge and architecture representations, and other domain-specific refinement knowledge that relates the scheduling domain to general algorithm design and data type refinement.

Using Planware, the user interactively specifies a problem and then the system automatically generates a formal specification and refines it. Various transportation schedulers have been generated, and with satisfactory performance.

## 1. Introduction

This paper presents an overview of Planware, a generator of high-performance scheduling algorithms, currently being developed at Kestrel Institute. Our aim is to convey a sense of the rationale for Planware, the design process that it supports, the architecture of the current Planware system, and our results to date. The reader may find more detail in the references.

Architecturally, Planware is an extension of the Specware system [SJ95], a system for developing formal specifications and refinements based on concepts from higher-order logic and category theory. Planware and Specware embody theoretical developments stemming from Kestrel's experience with previous systems, such as KIDS [Smith90a] and DTRE [BG91].

The goal of Planware is to allow experts in planning and scheduling to assemble quickly a specification of a scheduling problem, and to generate automatically a high-performance scheduler from it. The user's interactions with the system are designed to be entirely in the scheduling domain -- the user does not need to read or write formal specifications, nor to understand the logical and category-theoretic foundations of the system. We have invested substantial effort in automating the construction of scheduling domain theories.

To assemble a requirement specification and underlying domain theory, Planware requires very little information from the user:

- to select from a menu various attributes that specify the tasks that need to be scheduled, and
- to select from a taxonomy of resource theories the particular kind of resource against which to schedule the tasks.

From this minimal amount of information, Planware can automatically

- generate a formal specification of the scheduling problem (plus the relevant background concepts that comprise a domain theory),
- reformulate the specification using datatype refinements to build some of the problem constraints directly into the schedule datatype, allowing a dramatic simplification of the specification,
- apply domain-independent knowledge about designing global search algorithms with constraint propagation,
- apply datatype refinements and optimization techniques, and finally
- generate Common Lisp code.

For example, after design and refinement, the specification of transportation scheduling domains comprises about 10,000 lines of text of which about 3000 lines are the scheduling algorithm (the remainder consists of axioms and datatype operations that are not needed by the scheduler).

A key point here is that the high level of automation in Planware is achieved by applying domain-specific control (via a hand-built tactic) (1) to construct a problem specification and domain theory, and (2) to apply a series of domain-independent design theories and code-generation rules. The result is a fast, correct, executable scheduler automatically constructed from the user's description of a scheduling problem.

In the next section, we provide a brief introduction to the specification and refinement formalisms in Specware. In Section 3, we describe Planware by stepping through its design process, illustrating each step via the construction of a transportation scheduler.

## 2. Specware

Specware supports the modular construction of formal specifications and the stepwise and componentwise refinement of such specifications into executable code. Specware may be viewed as a visual interface to an abstract data type providing a suite of composition and transformation operators for building specifications, refinements, code modules, etc. This view has been realized in the system by directly implementing the formal foundations of Specware; category theory, sheaf theory, algebraic specification and general logics. The language of category theory results in a highly parameterized, robust, and extensible architecture that can scale to system-level software construction. A more detailed description of Specware may be found in[SJ95].

### 2.1. Specware concepts

A *specification* (or simply a spec or theory) defines a language and constrains its possible meanings via (higher-order) axioms and inference rules. A basic specification consists of a list of sorts, operations and axioms. For instance, the theory of partial orders can be presented as an abstract sort with a binary operation that satisfies the following properties: reflexivity, transitivity and antisymmetry. A Specware spec for this theory is:

```
spec PARTIAL-ORDER is
  sort E
  op leq: E, E -> boolean
    axiom transitivity-axiom is
      leq(x, y) & leq(y, z) => leq(x, z)
    axiom reflexivity-axiom is
     fa(x: E) leq(x, x)
    axiom anti-symmetry-axiom is
      leq(x, y) & leq(y, x) => x = y
  end-spec
```

Another example is a specification for a simple problem theory (DRO-SPEC) that consists of input domain, output range and a predicate that relates input to output.

```
spec DRO-SPEC is
  sort D, R
  op O: D, R -> boolean
 end-spec
```

Specifications can be used to express many kinds of software-related artifacts, including application domain theories, formal software requirements, abstract data types, abstract algorithms, and programming languages.

A *specification morphism* (or simply a spec-morphism or morphism) consists of two specs and one mapping, which maps the source spec to target spec via sorts and operations maps such that sorts map is compatible with operations map, and moreover, axioms in the source spec are theorems in the target spec. For instance, a spec-morphism from the partial-order theory to integer can be represented as:

```
spec-morphism INTEGER-IS-PARTIAL-ORDER:
 PARTIAL-ORDER -> INTEGER is
{E-> integer, leq -> <=}
```

Assuming that there is a spec SORTING-SPEC for the problem of sorting sequences of integers, a spec-morphism from DRO-SPEC to SORTING-SPEC can be expressed as:

```
spec-morphism DRO-to-SORTING: DRO-SPEC -> SORTING-SPEC is
{D -> set-of-integer, R -> sequence-of-integer, O -> sorting-pred}
```

where we assume that the sorting specification SORTING-SPEC has a predicate *sorting-pred* to specify sorting requirements.

Specification morphisms underlie several aspects of software development, including the binding of parameters in parameterized specifications, specification refinement and implementation, datatype refinement, and algorithm design [Smith90b].

An interpretation is a pair of spec-morphisms that essentially enables mapping an item to a term, which is what we need to express a refinement (or implementation) from one spec to another. Returning to our previous spec-morphism example with SORTING-SPEC, suppose that we do not have a predicate for sorting-predicate, then it is impossible to map O to any predicate symbol in SORTING-SPEC. However, we can map it to a term of SORTING-SPEC by forming a conjunction of all predicates that specify sorting requirements. This can be expressed via two spec-morphisms as follows:

> *interpretation* DRO-to-SORTING: **DRO-SPEC => SORTING-SPEC** *is*
> *mediator* **SORTING-SPEC-WITH-SORTING-PRED**
> *dom-to-med* {D -> *set-of-integer*, R -> *sequence-of-integer*, O -> *sorting-pred*}
> *cod-to-med import-morphism*

Here, we created a new spec SORTING-SPEC-WITH-SORTING-PRED which imports SORTING-SPEC and adds another predicate *sorting-pred* that is defined from the predicates in SORTING-SPEC. In the scheduling domain, a scheduling spec has normally a list of constraints (some of them are provided by users, and thus the list is dynamically constructed). To start the refinement process on a scheduling problem spec (DOMAIN-SPECIFIC-SCHEDULING), we need to construct an interpretation from DRO-SPEC to DOMAIN-SPECIFIC-SCHEDULING, which will be given in detail in section 3. It is indeed an interpretation since O in DRO-SPEC has to be mapped to the conjunction of all scheduling constraints present in DOMAIN-SPECIFIC-SCHEDULING. More precisely, an interpretation consists of two spec-morphisms: one from source spec to mediator spec, another from target spec to mediator spec that is required to be a definitional extension of target spec. In the following, we will use interpretation and refinement interchangeably.

Using spec-morphisms, structured specs can be constructed, via renaming and colimit. Colimit is the most basic way to put specs together. It requires a spec-diagram (a graph with nodes labeled by specs and arcs by spec-morphisms) called a base spec-diagram (also called a cover of that colimit/spec), and computes a shared union of that spec-diagram. Colimits are used intensively in the construction and factorization of the scheduling domain knowledge base in Planware

Given a spec, one refines it to more concrete specs via a sequence of refinements, so we need a sequential composition of interpretations to put these refinements together. Given a structured spec, for instance a spec formed via colimit, one only needs to give component interpretations of the cover, using parallel composition operator, a refinement for colimit object can be constructed automatically, provided the components interpretation are compatible with each other (this is where interpretation-morphism is used). Sequential and parallel compositions are used in the various Planware design tactics that will be described in section 3.

Finally, given an executable spec, code (Lisp or C++ in Specware system) can be generated. This is expressed via a Spec-to-Lisp interpretation or Spec-to-C++ interpretation. This work is based on general logics [Meseguer89] and their morphisms, with some modification. Our work in Planware of extending and applying Specware focuses basically on automating various combination of sequential and parallel compositions, and knowledge representations.

## 3. Planware Design Process

Planware aims to provide a framework that is general enough to allow the synthesis of schedulers in a wide range of domains. The key to achieving this generality was our development of a specification for a generic scheduling problem that can be refined into a variety of concrete scheduling problems. Our confidence in this abstract scheduling specification arises from experience with using the KIDS system to generate schedulers for such domains as transportation, manufacturing, power plant maintenance, satellite communications, pilot training, and others [SPW96].
Briefly, here is how the Planware design process works. The user is asked to supply information about a particular scheduling problem. This information is used to refine the abstract scheduling specification to a specification of the user's problem. Planware then applies tactics that automatically perform problem reformulation and simplification, algorithm design, datatype refinement, expression optimization, and finally code generation. The following sections describe the steps in the Planware design process in more detail.

The most time consuming and novel aspect of this work is the automatic construction of a domain theory for the particular scheduling problem. In the KIDS system, this construction typically required weeks or months of time. In Planware this time is reduced to minutes, but for a sharply restricted domain.

### 3.1. Abstract Scheduling Problems

Abstractly, we consider a scheduling problem to be a set of reservations, where each reservation consists of a start-time, tasks to be accomplished and resources allocated. We do not specify the tasks and resources in detail, since these may vary from one scheduling domain to another. We specify abstract scheduling problems as a function that takes a set of tasks and a set of resources as input, and returns a set of reservations (a schedule) that accomplishes all tasks and uses only provided resources.

In the current system, the abstract scheduling spec is limited to problems of scheduling a single (class of) resource; e.g. scheduling cargo on aircraft, or scheduling the duty periods of personnel. Our next challenge is extending the abstract scheduling spec to allow multiple classes of resource, and the constraints on their interactions.

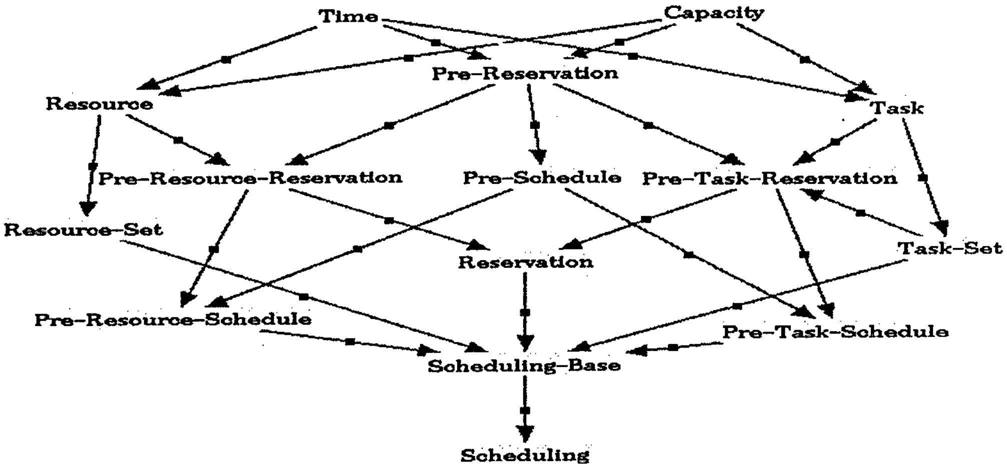
#### 3.1.1. Abstract Scheduling Specification

The specification of the abstract scheduling problem is structured as a diagram, with component specifications as nodes, and morphisms that relate the components as arcs. This structuring buys us: reusability, extensibility, implementability and evolutionary support of (re-)design. Basically, Planware's abstract scheduling specification has the following components:

* Time: a unit that is an instance of total order,
* Capacity: an instance of total order with group structure,
* Pre-Reservation: an abstract base reservation with start-time,
* Pre-Schedule: a set of base abstract reservations,
* Resource: an abstract resource spec,
* Task: abstract task spec,
* Reservation: an abstract reservation with start-time, resource and tasks,

* Pre-Resource-Schedule: an abstract schedule for resource allocation,
* Pre-Task-Scheule: an abstract schedule that accomplishes tasks,
* Scheduling-Base: a set of abstract reservations,
* Scheduling: an abstract scheduler with resource and task constraints, that is, all tasks are scheduled using only the provided resources.

The above version is, however, a simplified description of the actual spec in the system. For instance, we actually divide Pre-reservation into Pre-time-reservation and Pre-capacity-reservation, and put them together (via colimit) to form Pre-reservation. Since they are basic components, all other structure will be complicated by this. The following spec-diagram shows their dependencies.



Some of the specifications in this diagram are presented below. Note how the attributes of *Task* and *Resource* are expressed as functions on those sorts (e.g. max-capacity). As new attributes are added under user guidance, we simply add new function symbols to the spec. Eventually, Planware refines *Task* and *Resource* to tuples and their attributes to projection functions.

```
spec RESOURCE is
  import TIME, CAPACITY
  sort Resource
  op max-capacity: Resource -> Quantity
 end-spec

spec TASK is
 import TIME, CAPACITY
 sorts Task
 op task-demand: Task -> Quantity
 end-spec
```

```
  spec SCHEDULING is
   import SCHEDULING-BASE-EXTENDED
   op Only-Available-Resources-Used: Resource-Set, Schedule -> boolean
   definition of Only-Available-Resources-Used is
   axiom Only-Available-Resources-Used is
    Only-Available-Resources-Used(resource-set, valid-schedule) <=>
   in(a-reservation, valid-schedule) => in(asset(a-reservation), resource-set)
   end-definition

  op All-Tasks-Scheduled: Task-Set, Schedule -> boolean
   definition of All-Tasks-Scheduled is
   axiom All-Tasks-Scheduled is
   All-Tasks-Scheduled(task-set, valid-schedule) <=>
   in(task, task-set)  =>
    ex(a-reservation: Reservation)
     in(a-reservation, valid-schedule) & in(task, tasks(a-reservation))
  end-definition

  op Scheduler: Task-Set, Resource-Set -> Schedule
   axiom CONSTRAINING-SCHEDULER is
   Only-Available-Resources-Used(resource-set, Scheduler(task-set, resource-set))
   & All-Tasks-Scheduled(task-set, Scheduler(task-set, resource-set))
 end-spec
```

### 3.1.2. Refining to a particular scheduling problem

Given the abstract scheduling specification, the very first step of refining to a given scheduling problem is to get information from the user about how to refine the resource and task components. For instance, a task in the user's problem may have a release-date and a due-date; if it is a transportation task then it may have an origin and destination. Currently Planware provides a taxonomy of task attribute specs for the user to select from. This taxonomy is straightforward to extend. Analogously, we have developed a taxonomy of resource theories which the user selects from. The next step is to obtain constraints on the scheduler from the user's choices.

A key goal of Planware is to free the user from the need to read or write formal specifications. To achieve this, we needed to find a way to lift information about tasks and resources into constraints on a scheduler. We observed that all of the constraints on tasks that we have dealt with can be characterized abstractly by means of a partial order. Intuitively, a feasible schedule of reservations must provide enough resource to meet the demand of the input tasks. This notion of meeting task demand particularizes to a partial order on each task attribute. For example, a due-date attribute on a task requires that the finish-time of its reservation be before the task's due-date (i.e. finish-time <= due-date). For another example, the sum of the weights of the cargo items in a transportation reservation must not exceed the max-capacity of the transportation vehicle. Given partial order information about a task attribute, it is easy to create a constraint over an entire schedule; for example, returning to due-dates; if schedule is the output of Scheduler(Tasks, Resources) then

*fa*(a-task: Task, a-reservation: Reservation, valid-schedule: Schedule)
in(a-reservation, valid-schedule) & in(a-task, tasks(a-reservation))
=> finish-time(a-reservation) <= due-date(a-task)

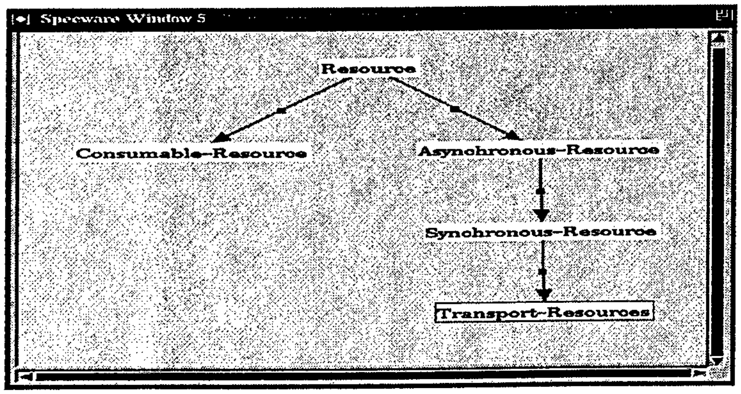
In fact, we require that a task attribute be not only partially ordered, but that it have greatest lower bounds (i.e. be a meet semi-lattice). This requirement comes from the needs of algorithm design -- the global search/constraint propagation algorithms perform fixpoint iteration in a semilattice.

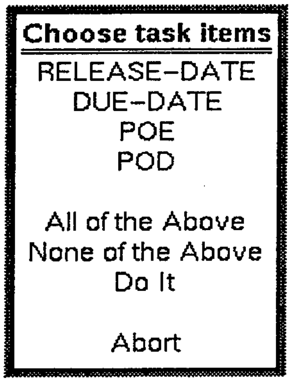
By restricting to semi-lattice-structured task attributes, the task of constructing a formal specification of a scheduling problem, which is usually tedious and error-prone, is simplified to just asking the user to input/select whether each attribute is a lower/exact/upper bound. The corresponding constraints are constructed and asserted as output conditions of the desired scheduler. Additional work is required to add in the appropriate constructors and other operators for the refined datatypes of *Task, Resource, Reservation,* and *Schedule.* At this stage, Planware also constructs a slightly weakened version of the reservation and schedule specs, called *Partial-reservation* and *Partial-schedule.* These form the basis for the global search algorithm designed in a subsequent stage.

### 3.1.3. Example -- Transportation Scheduling

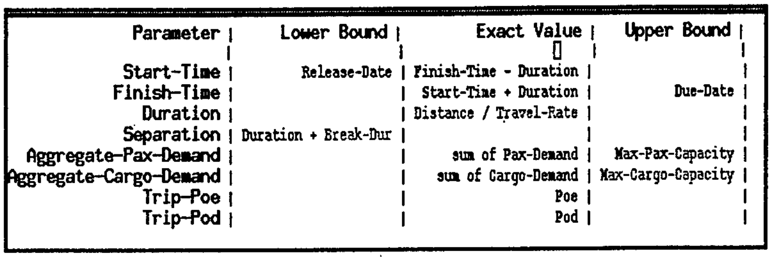
In a simple transportation scheduling problem, the input tasks are movement requirements, which are descriptions of cargo that have to be moved. In this simple version a movement requirement includes information about when the cargo is available to be moved and by when it must arrive. So a schedule is a set of trips. Each trip has a start time and a manifest -- the set of movement requirements that it has been assigned to execute.

The first phase of our development is to construct a transportation scheduling specification. Suppose we have enriched our resource taxonomy and task taxonomy to allow us to have basic transportation domain information like release-date and due-date, as well as the origin and destination of a trip, which are expressed by key words POD and POE, respectively. The following figure shows user selection interface.

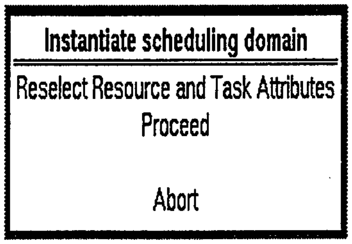




Suppose we have selected transportation resource and all task attributes as shown in the above figure. Here, the colimit operator is used to put all task attributes together to form a domain-specific task spec. The parallel composition operator is used to put all domain-specific interpretations together to form a domain-specific scheduling spec. The constraints present in this selection are summarized below.



| Parameter | Lower Bound | Exact Value | Upper Bound |
|---|---|---|---|
| | | □ | |
| Start-Time | Release-Date | Finish-Time - Duration | |
| Finish-Time | | Start-Time + Duration | Due-Date |
| Duration | | Distance / Travel-Rate | |
| Separation | Duration + Break-Dur | | |
| Aggregate-Pax-Demand | | sum of Pax-Demand | Max-Pax-Capacity |
| Aggregate-Cargo-Demand | | sum of Cargo-Demand | Max-Cargo-Capacity |
| Trip-Poe | | Poe | |
| Trip-Pod | | Pod | |


Instantiate scheduling domain
Reselect Resource and Task Attributes
Proceed
Abort

The next step in the development process, if we choose to go forward, automatically constructs a transportation scheduling specification via the tactics described above.

### 3.2. Data-Type Reformulation

The construction process described above produces a scheduling specification for a particular problem. It is still formulated in terms of the schedule datatype which is a set of reservations. This formulation is general and supports the initial problem acquisition stage in Planware, but it is a relatively poor datatype for implementation purposes. In this stage, the Planware design process applies a datatype refinement that is stored with the resource theory that was chosen from the resource taxonomy. The effect is to refine *Schedule = set(Reservation)* into a datatype that is better suited to the resource properties. The payoff is that we can then simplify away some of the problem constraints because they are effectively built into the schedule datatype. After Planware refines the schedule datatype, it invokes a context-dependent simplification tactic [Smith90a] to simplify the constraints.

**Example -- Transportation Scheduling**

After finishing the refinement of abstract scheduling to transportation-scheduling, a series of refinements is carried out: adding complete constructors for transportation schedule data type; refining set of reservations to a map that maps a resource to its scheduled tasks (in sequence with increasingly start-time as ordering); etc.

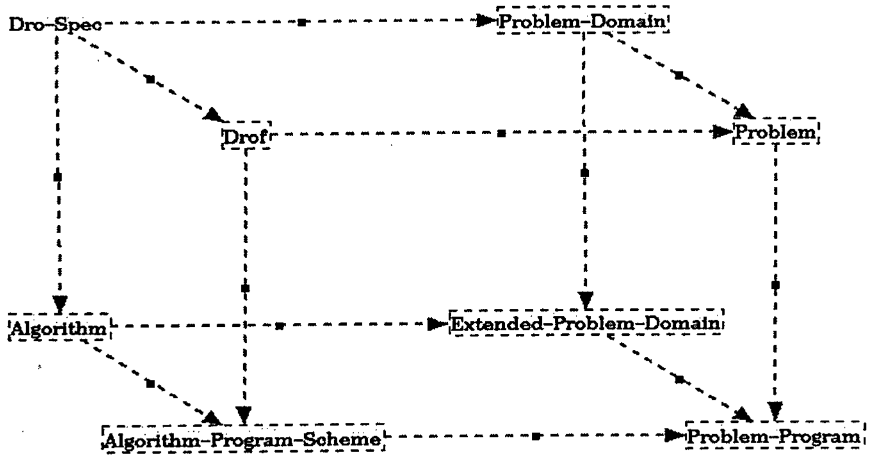
The transportation scheduling problem uses a transportation resource which is a refinement of a synchronous resource (i.e. all reservations on a synchronous resource must be synchronized in the sense that two reservations must be either separated in time by at least some minimal amount or else simultaneous -- starting and ending at the same time). Planware has refinements from *set(Reservation)* to *map(Resource, seq(Trip))* which effectively implements a schedule as an itinerary -- for each resource we have the sequence of trips that it makes. The characteristic synchronization constraint is then simplified from a complex disjunction to a simple linear check over adjacent trips. For a typical input of 10,000 movement requirements, the original formulation will have several hundred millions ground disjuncts for the synchronization constraint, versus about 100,000 in the refined formulation.

### 3.3. Algorithm Design

A design theory for an algorithmic concept can be represented as a formal specification [SL90].

Any particular instance of that design theory corresponds to an interpretation from it to a specification of the particular problem being solved. For instance, various interpretations from divide-and-conquer theory to a sorting specification correspond to various sorting algorithms, such as quicksort, mergesort or Batcher's sort. Design theories can be arranged in a refinement hierarchy with specification morphisms providing the refinement links; e.g. a hierarchy of algorithm theories is presented in [Smith96]. The concepts and procedures described below are intended to automate the process of algorithm design by choosing a chain of algorithm design theories for a particular problem, and construct an interpretation from the chosen design theory to that problem. Thus, an algorithm for the specific problem is constructed.

The representation of our algorithm design framework can be illustrated by the following diagram, let us call it algorithm design cube, or simply the cube in this paper. The arrows in the cube represent the relationship between abstract theory and the concrete problem. Technically, the left square in the cube is a spec-diagram corresponding to the abstract algorithm design knowledge; the right square in the cube corresponds to the domain-specific problem and program scheme. The arrows in between are interpretations. Essentially, design tactics described below are based on sequential and parallel refinement composition operators, as well as others. In the following the intended meaning of each arrow (and spec) and the way to construct them is described in detail.



At the very beginning, we have only the node DRO labeled with the abstract problem domain theory, as it can be seen in the above algorithm design cube. When a concrete (or domain-specific) problem specification is chosen (along with the main function to be developed), a problem domain specification can be extracted from it (and it is done via an extract tactic that gives a spec-morphism as result). So, the upper morphism on the right side of the cube is constructed.

The second arrow construction tactic, the domain-specific interpretation tactic, is a little more complex. The domain-specific interpretation tactic works as follows, first, use the main function signature to construct an interpretation from DRO to the problem domain specification. Second, use DRO and the main function signature, choose a DROF spec from the possible solutions specifications, e.g. all solutions spec, one solution spec and optimal solution spec, and maybe other kind of solution specs if exist. Third, compute the colimit of the spec-diagram that include DRO, the domain-specific problem domain specification and DROF, which gives, among others, an interpretation from DROF to the colimit object. Finally, we check that the computed colimit is isormorphic to the domain-specific problem specification. In doing so, we have constructed and constructively proved that the base diagram of the computed colimit, namely, DRO, DROF and the problem domain specification is a cover of the domain-specific problem specification. Informally speaking, we can use DRO, DRFO, and the problem domain specification to construct a program scheme, and that will be a program scheme for our specific problem too.

The third arrow construction tactic is called classification and it involves a process called ladder construction (see [Smith96] for details). Here, we only give a brief description of it in the context of algorithm design. Basically, this tactic consists of two steps: (1) selecting an appropriate design theory from a refinement hierarchy of design theories, and (2) constructing an interpretation. The first step is in general interactive, but can be automatic in certain domains (e.g. scheduling domains). The second step is accomplished via the ladder construction process as described below.

$$\begin{tikzcd}
DT_0 \arrow[r, Rightarrow, "I_0"] \arrow[d] & Spec_0 \arrow[d] \\
DT_1 \arrow[r, Rightarrow, "I_1"] \arrow[d] & Spec_1 \arrow[d] \\
DT_2 \arrow[r, Rightarrow, "I_2"] \arrow[d] & Spec_2 \arrow[d] \\
\vdots & \vdots \\
DT_n \arrow[r, Rightarrow, "I_n"] & Spec_n
\end{tikzcd}$$

Ladder Construction

The process of incrementally constructing an interpretation is illustrated in the ladder construction diagram to the left. The left-hand side of the ladder is a path in a refinement hierarchy of design theories starting at the root. The ladder is constructed a rung at a time from the top down. The initial interpretation from problem theory to Spec0 may be simple to construct. Subsequent rungs of the ladder are constructed by a constraint solving process that involves user choices, the propagation of consistency constraints, calculation of colimits, and constructive theorem proving like witness finding and unskolemization as described in detail in [Smith93] [Smith96].

The result of the classification and ladder construction tactic is a concrete algorithm design theory and its program scheme, and an interpretation from algorithm design theory to the problem domain.

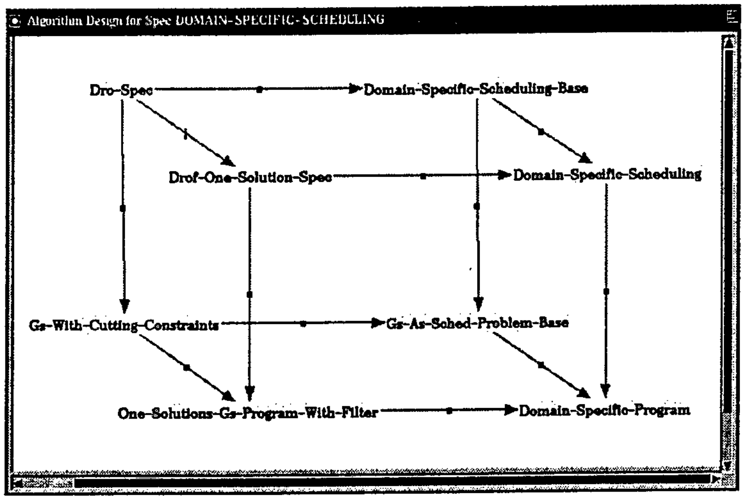
The last tactic, the program scheme instantiation tactic, computes a colimit of the diagram that consists of algorithm design theory, its program scheme and the extended problem domain. The colimit object is domain-specific program scheme. Last but not the least, there must be a specification morphism from domain-specific problem spec to the constructed program. This is constructed and constructively proved to always exist by universal property of colimits.

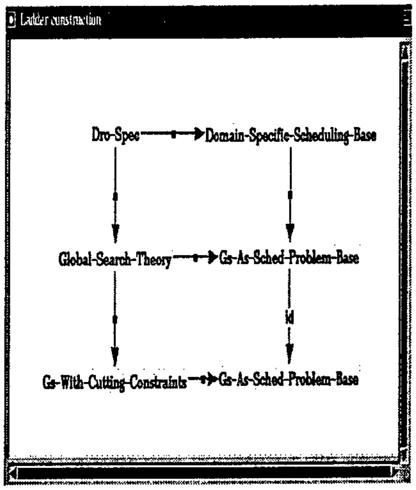
With these four tactics, given a concrete problem, we can semi-automatically construct a program theory for that problem based on the selected and successfully interpreted algorithm design theory. Normally, further steps are needed to make it executable or more efficient.

The first design theory used in Planware is global-search theory and its extension with cutting constraints. Since this decision is fixed it is applied with no need for further interaction. Another algorithm design tactic used is constraint propagation. This amounts to generating basic constraint propagation procedures given a kind of scheduling problem domain, and synthesizing domain-specific constraint propagation procedures after the instantiation phase. Basically, that amounts to generating constraint propagation for a set of upper bounds, exact bounds and lower bounds of domain-specific constraints. Technically, this is related to data type refinement to get the right constructors for each data type used in the constraints; and to the instantiation of corresponding semi-lattice structures. After getting all the constraints propagation procedures, they are composed together and a flat semi-lattice is constructed that consists of a tuple of all component semi-lattice. Notice that this can only be done dynamically, since the constraint structure varies from one domain to another.

### Example -- Transportation Scheduling

The result of algorithm design is shown in the following diagram, the bottom right node is the program scheme for transportation scheduling problem.


Algorithm Design for Spec DOMAIN-SPECIFIC-SCHEDULING
Dro-Spec
Domain-Specific-Scheduling-Base
Drof-One-Solution-Spec
Domain-Specific-Scheduling
Gs-With-Cutting-Constraints
Gs-As-Sched-Problem-Base
One-Solutions-Gs-Program-With-Filter
Domain-Specific-Program



The constraints present in this scheduling problem are cargo and pax capacity constraints, release and due date constraints, and trip origin and destination constraints, as well as trip separation constraints.

## 3.4. Expression Optimization

This stage will apply various expression optimization refinements, such as context-dependent simplification, common-subexpression elimination, finite differencing, partial evaluation, and so on. These are not currently applied in Planware.

## 3.5. Automatic Code Generation

We have developed a code-generation tactic that automatically generates code for a structured spec, provided the structure is (recursively) of the following form:

* directly implementable,
* definitional extension or a translation of an implementable spec,
* colimit, each component of which is implementable (recursively),
* instantiation of implementable specs,
* can be interpreted to an implementable spec.

If there are multiple choices, we use a heuristic to decide which way to go. In the Planware context, given the scheduling system architecture, we can generate code for it if each instantiated component is implementable, and we further specialize the code-generation tactic by a specific implementation order imposed by the dependencies of the scheduling systems architecture.

## 4. Concluding Remarks and Future Work

We have presented our Planware system for generating domain-specific high-performance scheduling software in a highly automatic way. Planware is an extension of the Specware formal development environment. Scheduling domain knowledge has been represented abstractly and structurally to enable user problems to be solved with minimal interaction. In particular, the resource and task taxonomies which specify general/domain-specific scheduling knowledge have been developed as well as their architectural relationship with the scheduling system architecture. For synthesizing domain-specific schedulers, a set of design tactics for instantiation to the concerned problem, data-type refinement, algorithm design with constraints propagation and automatic code-generation have been developed and successfully applied. We have experimented with the transportation scheduling domain and developed various schedulers there with satisfactory performance.

We believe that Planware is a new paradigm for domain-specific software generators. Planware differs from other domain-specific software generators in that it is built on a foundation of domain-independent general-purpose software specification and synthesis capabilities (Specware/ Designware). In particular, Planware relies on (1) the Specware capabilities for composing specifications, refining them and translating to code; and (2) the Designware libraries of domain-independent design knowledge about algorithms, datatype refinements, and expression optimization techniques (and their application tactics) to construct refinements. The domain-specificity of Planware comes in the form of (1) specifications of domain knowledge in the form of the abstract scheduling specification, the taxonomies of task and resource theories, etc. and (2) scheduling-specific tactics for controlling the Planware design process; e.g. tactics for lifting properties of tasks to constraints on the scheduler, tactics for lifting resource constraints to scheduling constraints, tactics for constructing the constructors and other datatype operations needed by the refined *Task, Resource, Reservation*, and *Schedule* specs, tactics for generating a global search theory for the problem at hand, etc. The background of domain-independent design knowledge allows a user to derive software even when the requirements fall outside the domain-specific scope of the system. The user then gets less automation, and must supply more guidance in the construction process.

There are many things to be done before Planware can be deployed. One crucial extension is allowing the user more flexibility in supplying task information. The spreadsheet-like display in Section 3 is a first prototype of our next interface. The user's choice of resource theory informs the parameter list and the plausible options for lower/exact/upper bounds on parameters. We are working to let the user choose and modify arbitrary entries. As before the user only interacts with the system in domain-specific terms. Another vital extension is to generalize the abstract scheduling spec to multiple resource classes. Another extension that is underway is to extend Planware to allow the synthesis of scheduling systems, including visual displays, editors, GUI, database mediators, and so on.

*Acknowledgment*

We would like to thank Cordell Green, Jim Mcdonald, T.C. Wang, David Espinosa, Richard Jullig and Y.V. Srinivas for discussions and suggestions during various stages of the Planware project. This project has been mainly supported by DARPA/Rome Lab under Contract F30602-95-C-0247 and by Rome Lab under Contract F30602-95-C-0036.

## 5. References

- [BG91] Blaine, L., and Goldberg, A. DTRE - a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Moeller, Ed. North-Holland, Amsterdam, 1991, pp. 165-204.
- [Meseguer89] Meseguer, J. General Logics, In *Logic Colloquium 87*, H.D. Ebbinghaus et al, Ed. North Holland, Amsterdam, 1989.
- [Smith90a] Smith, D. R. KIDS - a semi-automatic program development system. IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16, 9 (September 1990), 1024-1043.
- [Smith90b] Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. In Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375, L. van de Snepscheut, Ed. Springer-Verlag, Berlin, 1989, pp. 379-398. (reprinted in Science of Computer Programming, 14(2-3), October 1990, pp. 305-321).
- [Smith93] Smith, D. R. Constructing specification morphisms. Journal of Symbolic Computation, Special Issue on Automatic Programming 15, 5-6 (May-June 1993), 571-606.
- [Smith96] Smith, D. R. Toward a Classification Approach to Design, in *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96*, LNCS 1101, Springer-Verlag, 1996, 62-84.
- [SPW96] Douglas R. Smith, Eduardo A. Parra, and Stephen J. Westfold, Synthesis of Planning and Scheduling Software, in *Advanced Planning Technology*, (Ed. A. Tate), AAAI Press, Menlo Park, California, 1996, 226-234.
- [SJ95] Srinivas, Y. V., and Juellig, R. Specware$^{tm}$: formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. Springer-Verlag, Berlin, 1995. Lecture Notes in Computer Science, Vol. 947.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-81056

# *MISSION*
# *OF*
# *AFRL/INFORMATION DIRECTORATE (IF)*

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.