Planware II:
Synthesis of Schedulers for Complex Resource Systems

Marcel Becker
Limei Gilham
Douglas R. Smith
Kestrel Technology
{becker,gilham,smith}@kt-llc.com

Abstract
Planware is an integrated development environment for the domain of complex planning and scheduling systems. Its design and implementation aim at supporting the entire planning and scheduling process including domain analysis and knowledge acquisition; application development and testing; and mixed-initiative, human-in-the-loop, plan and schedule computation. Based on principles of automatic software synthesis, Planware addresses the problem of maintaining the synchronization between evolving specifications, and the corresponding system implementation. Planware automatically generates optimized and specialized planning and scheduling code from high-level models of complex problems.

Resources and tasks are uniformly modeled using a hierarchical state machine formalism that represents activities as states, and includes constructs for expressing constraints on states and transitions. The generator analyzes the state machine models to instantiate program schemas generating concrete implementations of backtrack search and constraint propagation algorithms. Coordination between resources and tasks is achieved through the use of services: tasks *require* services, and resources *provide* services. Planware's scheduler generator component matches providers with requesters, and automatically generates the code necessary to verify and enforce, at schedule computation time, the service constraints imposed in the model. Planware's user interface is based on Sun's NetBeans platform and provides integrated graphic and text editors for modeling complex resource systems, automatically generating batch schedulers, and executing the generated schedulers on test data sets.

1. Introduction

Planware is an integrated development environment for modeling scheduling problems, and automatically generating high-performance batch scheduler systems. Planware supports

1. The precise modeling of complex resources and their interactions,
2. The automatic generation of high performance scheduling components from those models,
3. The synthesis of application-specific tools to help the user visualize complex schedules
4. XML-based glue to support the integration of generated schedulers into target environments.

The main innovation of this work is a state machine-based formalism for modeling complex resource systems. This formalism provides a good balance between abstraction (minimal implementation detail), precision (clear semantics), and expressiveness (supporting the modeling of a wide range of planning, scheduling, and resource allocation problems). The key to modeling complex resource systems is to represent single resources together with their interaction. The dynamics of both tasks and resources are represented by state machine-based models. A key insight was that the interaction among resources could be modeled by extending the state machine formalism to support *services*. The service description provides a very powerful mechanism to specify dependencies and constraints between resources using a simple and intuitive syntax. For example, cargo is modeled as an activity that requires transportation service. An aircraft provides transportation service and, in turn, requires the service of a crew. The matching of the required and offered services in this case prescribes the need for a synchronized reservation of the cargo, the aircraft, and the crew for each flight segment.

The Planware formalism is a specification language, not a programming language. A model of a complex resource system provides no intrinsic clues about how to find schedules that satisfy its constraints. Algorithmic and data structuring knowledge is added by the generator to produce the scheduler code. In this sense Planware is related to systems such as Amphion [Lowry94] and AutoBayes [Fischer00] that provide automatic code generators for domain-specific specification languages.

2. Modeling Complex Resource Systems

Planware's main goal is to automatically generate high-performance resource planners and schedulers for problem domains involving large numbers of different resources interacting according to complex sets of constraints. Traditional approaches to modeling complex scheduling problems, like the ones based on mathematical programming techniques, typically result in a large monolithic model that is difficult to solve, understand, and maintain. To avoid this drawback, Planware takes a compositional approach to modeling with the objective of minimizing model complexity, and facilitating model maintenance and understanding. Some of the requirements for the modeling formalism are:

1. Should naturally support the concepts defined by a general planning and scheduling ontology.
2. Should map to a graphical representation of models capable of conveying a high level view of the problem structure.
3. Should use syntax constructs familiar to most programmers and computer scientists.
4. Should use a modeling paradigm similar to the ones used by modern computer languages.
5. Should separate domain knowledge from problem-solving knowledge.
6. Should minimize modeling complexity by embedding complex constraints in the internal structure (implicit semantic) of models.
7. Should represent tasks and resources using the same formalism.
8. Should represent interactions between resources/tasks and associated constraints.
9. Should distinguish between individual resource constraints and constraints resulting from interactions among different resources.
10. Should implicitly convey the notion of elapsed time, duration of activities, and temporal dependencies.

The formalism developed for Planware, based on an extension of Abstract State Machines [Gurevich, Pavlovic02], addresses all the requirements above. In the next paragraphs we describe how abstract state machine models are used to represent resources and tasks. Before presenting the actual structure and syntax of the models, we describe the intuition that drove the development of the formalism.

Resources as Activity machines

[Smith97] describes an ontology for planning and scheduling systems. The five top-level entities in this ontology are *tasks* or *demands*, *activities*, *resources*, *services*, and *constraints*. Using this ontology, the role of a scheduling or planning system can be described as the prescription of a sequence of activities that a set of resources must perform over time to perform the services required by a task. Based on this description, it is clear that any formalism for describing a scheduling problem domain must be able to represent tasks, resources, activities, and services, plus associated constraints.

If we consider the processing of an activity by a resource as a possible "state" or "mode" the resource entity can assume, we can think of a resource model as the description of all the valid sequences of activities it can perform. For example, a transportation aircraft might have the following sequence of activities:

Prepare → fly → land → fly → fly → unload → fuel → fly → and so on

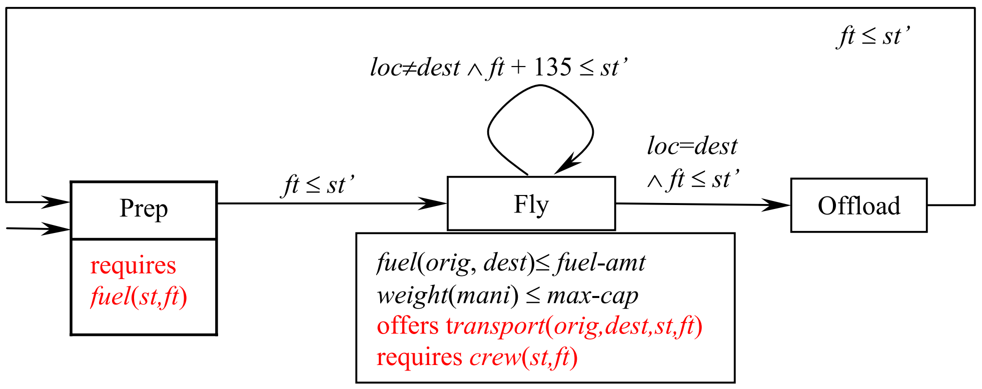
Or

Prepare → fly → fly → unload → fuel → fly → and so on

Each legal sequence of activities is called a *behavior*. A resource is characterized by a potentially infinite collection of behaviors. The usual trick for concisely representing an infinite collection of behaviors is by a state machine. Our modeling approach is based on state machines, although the term "state" is somewhat misleading and we prefer to call our models *activity machines*, as exemplified in Figure 1.

In the activity machine diagram, we refer to the boxes/states as *activities* and the arrows are called *transitions* and indicate which activities can legally follow one another.

The activity sequences per se give us general information about the sequencing of activities for a resource, but little information about the activities. Clearly in scheduling we need to model the timing of activities, as well as



$ft \leq st'$
$loc \neq dest \wedge ft + 135 \leq st'$
$loc = dest$
$\wedge ft \leq st'$
$ft \leq st'$
Prep
Fly
Offload
requires
fuel(*st*,*ft*)
fuel(*orig*, *dest*)≤ *fuel-amt*
weight(*mani*) ≤ *max-cap*
offers *transport*(*orig*,*dest*,*st*,*ft*)
requires *crew*(*st*,*ft*)

Figure 1 Activity Machine For Transportation Aircraft

capacity of a resource to do useful work, and other physical constraints. The next step in modeling resources is to represent information about an activity via activity variables, simply called *variables*. The set of variables defined for a particular resource model represents a state descriptor that is used to describe all activities performed by this resource. For example, a Fly activity of a transportation resource might be characterized by variables that model its *start time*, *finish time*, *origin*, *destination*, and others. Planware uses the same collection of variables to characterize all activities of an activity machine, and this collection is sometimes called the signature of the machine (See Figure 2). Technically, each activity is treated as a first-order theory, with the signature providing the vocabulary of the theory, and the axioms specifying constraints on the meaning of the variables (and other vocabulary).

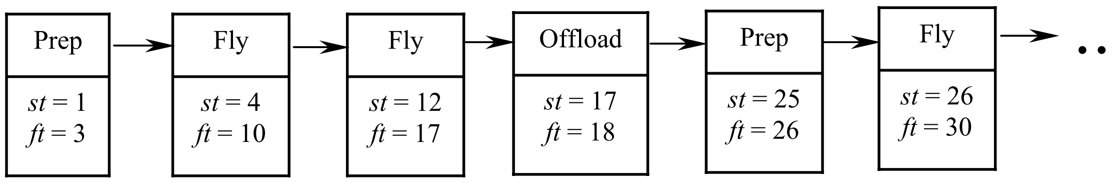
The variables defining the signature of an activity machine are further divided into four groups: *constants, internal variables, external variables*, and *input variables*.

- **Constants** are the fixed parameters used to characterize invariant aspects of the resource. For example, the size of the cargo holder of an aircraft, its maximum speed, the maximum amount of fuel it can carry are parameters that can be constants for a given aircraft type. The set of constants define the input values that should be provided to the scheduling system to create concrete instances of resources at schedule computation time.
- **Internal variables** are auxiliary variables used by the scheduler for bookkeeping purposes. For example, if there is a constraint that states than an aircraft needs to go through preventive maintenance after a certain number of hours flown, an internal variable can be used to maintain the number of hours flown since last maintenance.
- **External variables** represent the values that are modified by the scheduler engine while computing the schedule. For example, the start and end time of an aircraft Fly activity will be set by the scheduler based on the availability of additional resources like Crews and Airports. External variable values depend not only upon the evolution of the resource machine but also upon the scheduling decisions made by the problem-solving control strategy.
- **Input variables** serve as a mechanism for passing parameters between a resource requesting a service and a resource providing it. For example, the origin and destination of a Fly activity may be specified as input variables whose values get assigned each time a transportation task needs to be satisfied. Input variables act as constant variables for a given fragment of the resource behavior. The role of the input variables and external variables will be further discussed when we present the concept of services and service matches.

Figure 2 Signature for ActivityMachine

constant *velocity*	:Velocity
constant *maxCargoCapacity*	:Capacity
internal-variable *duration*	:Duration
input-variable *requiredCargoCapacity*	:Capacity
input-variable *Origin*	:Location
input-variable *Destination*	:Location
external-variable *startTime*	:Time
external-variable *endTime*	:Time

Now, with variables that take on possibly new values at each activity, we can express more information about a behavior:



Most variables of interest hold their values for the duration of an activity, and only change with the transition between activities. In the syntax construct used to describe valid transitions, the variables whose values change as a result of the transition are explicitly represented. Three external variables are pre-defined for all activity machines, and do not need to be explicitly represented in the assignments field of a transition: *start-time, end-time*, and *duration*. It is implicit in the structure of the activity machine the constraint that states that the start-time of an activity is always greater than or equal to the end-time of the preceding activity; and that the end-time of an activity is equal to or greater than the sum of its start-time and its duration.

Constraints

Not all combinations of values for the variables are physically possible. For example, if the capacity of an aircraft is 100 tons, then an activity in which the *mani-wgt* variable has a value exceeding 100 tons doesn't model a realizable situation. To rule out such impossible situations, each activity has *axioms* that express constraints on the values that variables can take on in an activity. For example, the Fly mode in Figure 1 rules out flying activities in which the aircraft is carrying too much weight, or too little fuel.

Furthermore, it is necessary to put constraints on transitions too, to model the physically realizable evolution of variables between activities. For example, the transition from Prep to Fly in Figure 1 specifies that the end time of activity Prep should less than or equal to the start time of activity Fly. The constraints labeling a transition must refer to the values of variables in both the before and after modes. The usual notation is to refer to the value of a variable x in the after state by priming it: x'. So the constraint $ft \leq st'$ means that the finish time of the before activity must be no later than the start of the after activity. As previously mentioned, a number of temporal constraints between modes do not need to be explicitly represented since the structure of the machine already assumes a number of temporal dependencies between the sequence of valid activities.

Services

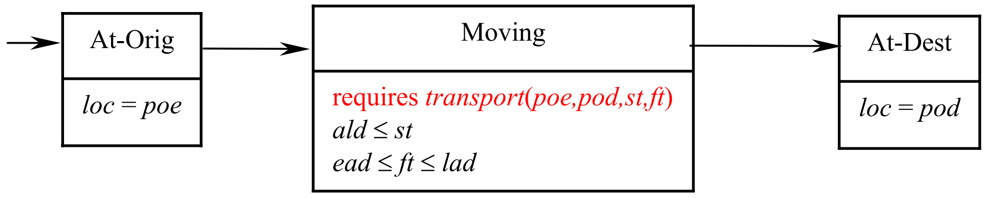
The modes or activities, the variables, the transitions, and the constraints are sufficient to represent the behavior of an individual resource. The key missing element of this formalism is how to connect resources to tasks, and how to coordinate the usage of several resources to accomplish complex tasks. For example, the transportation of certain amount of cargo between two locations may involve the usage of a number of different aircraft, airports, crews, fuel, ground control personnel, diplomatic clearances, etc. We need to provide modeling constructs that allow the explicit representation of these dependencies in our model. The missing modeling construct is the *service*. The service is the element used to coordinate and synchronize the execution of activities across different resources. Each activity machine may specify required and/or provided services. Machines that only request services define the top-level tasks that drive the scheduling process. Resources are machines that provide one or more services. Resources can also request additional services. For example, to provide transportation service to a transportation request, the aircraft may need services from one or more crew resources. In this case, the resource plays the dual role of provider and requester. The concept of *requested* and *provided* services allows task and resources to be represented using a uniform formalism.

A service is specified by a predicate associated with a mode together with an indication of whether it is a provided or required service. For example, transportation service may be represented by a predicate *Transport*(*orig, dest, st, ft*) which specifies a transportation activity from an origin *orig* to a destination *dest* beginning at time *st* and ending at time *ft*. The following mode machine provides a simple model of a transportation aircraft.

The requester resource specifies the service as a required condition. The provider specifies it as a provided condition. For temporal synchronization, a service can be specified as a pre-condition, a post-condition, or an invariant. If the service is specified as an invariant, both activities, the requesting and the providing, should start and end at the same time. For the other types, there are set of rules to establish the appropriate synchronization depending on the characteristics of the provider and requester. For example, if the requesting service is a pre-condition and the providing service is a post-condition, the providing activity should finish before the requesting activity can start.

There is also a set of rules governing the assignment of values to the parameters specified in the service description. For the requesting resource, external variables present in the service predicate will have their values set by the scheduler after an appropriate provider has been identified. All other variables will not change values. For the providing resource, input variables present in the service predicate will act as constants for the purpose of finding a valid sequence of activities to satisfy the request. External variables will be unified with external variables coming from the requester. At scheduling time, any constraints imposed on the external variables of the requester, will be translated to the corresponding variables of the provider.

Before we go on to discuss how services between machines are linked up, we note that the introduction of services in mode machines allows us to treat tasks as a special case of resource. Tasks (or goals or activities) are the drivers of a planning or scheduling problem. The overall nature of the scheduling problem is to carry out a set of tasks subject to the constraints imposed by the available resources. In terms of our mode machine model, a task can be modeled as a resource that requires service, but offers none. For example, a transportation task might be modeled as a simple machine with three modes:



with signature:

constant *poe, pod*	: Location
constant *ald, ead, lad*	: Time
constant *demand*	: Capacity
external-variable *st,ft*	: Time
internal-variable *duration*	: Duration

The first mode represents the cargo waiting to be transported at its origin, called *poe* (port of embarkation). The second mode represents the transportation activity during which transportation service is required. And finally, the mode machine has a final/accepting mode in which the cargo has arrived at its destination, called *pod* (port of debarkation).

Note how the service requirement is given two constants (*poe* and *pod*) rather than variables. The reason of course is that the task provides this data to the resource that will carry out the transporting. Additionally, note that the Moving mode has two axioms which express constraints on the start and finish time of the Moving activity — the start time must occur no earlier than the *ald* (Available-to-load) time and the finish time must occur between the *ead* and *lad* times (earlier-arrival-date and latest-arrival-date resp.) In other words, the service requirement can be thought of as a service request, complete with whatever data and constraints are relevant.

What is crucial here is that we now have various mode machines each of which governs the behavior of a class of tasks or resources. We have modeled the component tasks and resources individually, and now we need to model the composite system. To model a complex resource system we focus on the interactions of the components, which are specified by the services.

We use the following *service match* formula schema to express the conditions under which the service provided by resource *Prov* satisfies the service required by resource *Req:*

$$
\begin{aligned}
&\forall(\mathit{constants}\ (\mathit{Req}),\ \mathit{input\text{-}vars}\ (\mathit{Req}),\ \mathit{constants}\ (\mathit{Prov}))\\
&\exists(\mathit{ext\text{-}vars}\ (\mathit{Req}),\ \mathit{internal\text{-}vars}\ (\mathit{Req}),\ \mathit{input\text{-}vars}\ (\mathit{Prov}),\ \mathit{ext\text{-}vars}\ (\mathit{Prov}),\ \mathit{internal\text{-}vars}\ (\mathit{Prov}))\\
&(\mathit{Provided\ Conditions}\ (\mathit{Req}) \wedge \mathit{ProvidedConditions}\ (\mathit{Prov})\\
\Rightarrow&\\
&\mathit{ReqConditions}\ (\mathit{Req}) \wedge \mathit{Constraints}\ (\mathit{Req}) \wedge \mathit{ReqConditions}\ (\mathit{Prov}) \wedge \mathit{Constraints}\ (\mathit{Prov}))
\end{aligned}
$$

While we reason about this formula statically, at design time, we do not actually expect the formula to be provable. We expect two kinds of information from reasoning about the formula. First, we get witnesses for the existentials, meaning that for each existentially quantified variable, we extract a term over the preceding universally quantified variables. Second, we gather up any of the conjuncts in the consequent of the formula that cannot be proved. These gathered constraints are the *aggregated constraints* of the composite resource *Req + Prov.* While they are not provable at design-time, they will be treated as constraints to enforce at run-time (i.e. schedule-generation-time), either via pruning or constraint propagation. The inference can be construed as a constructive version of directed inference [Smith82, Smith85].

For example, an instance of the schema for simplified models of cargo requirement and transportation aircraft is

$$
\begin{array}{l}
\forall\ (theMvr, POE, POD, ALD, LAD, EAD, theAc, FuelAmt, MaxCap) \\
\exists\ (ac, st_{mvr}, ft_{mvr}, mvr, orig, dest, st_{ac}, ft_{ac}, dur_{ac}) \\
\quad (inTransit\ (mvr, theAc, orig, dest, st_{ac}, ftac), \\
\Rightarrow \quad inTransit\ (theMvr, ac, POE, POD, st_{mvr}, ft_{mvr}) \\
\quad \wedge ALD \leq\ st_{mvr}\ \wedge EAD \leq ft_{mvr} \wedge ft_{mvr} \leq LAD, \\
\quad \wedge fuel\ (orig, dest) \leq FuelAmt \wedge weight\ (mani) \leq MaxCap)
\end{array}
$$

and yields the following substitutions (i.e. witness for the existentially quantified variables):

$$mvr := theMvr,\ ac := theAc,\ orig := POE,\ dest := POD,\ st_{mvr} := st_{ac},\ ft_{mvr} := ft_{ac}$$

plus the following derived constraints

$$
\begin{array}{l}
ALD \leq\ st_{ac} \wedge ALD \leq\ ft_{ac}\ \wedge\ ft_{ac} \leq LAD \wedge \\
fuel(POE, POD) \leq FuelAmt \wedge weight(mani) \leq MaxCap
\end{array}
$$

Note how the service match exchanges data between the requester and provider. Here, the cargo requirement supplies the start and end points as well as the key time parameters: ALD, EAD, LAD. The aircraft, on the other hand, provides its identifier as the resource that will carry out the task. The exchange of information also includes the merging of constraints.

In Planware the actual ground constraints are determined dynamically, and they depend on the input data together with the dynamics of the scheduling process (the current state of the process). This is in contrast to many Operations Research and Constraint Programming systems in which a static set of constraints is passed to a generic solver. Planware not only generates a customized solver for each problem, but that solver works on a dynamic constraint problem.

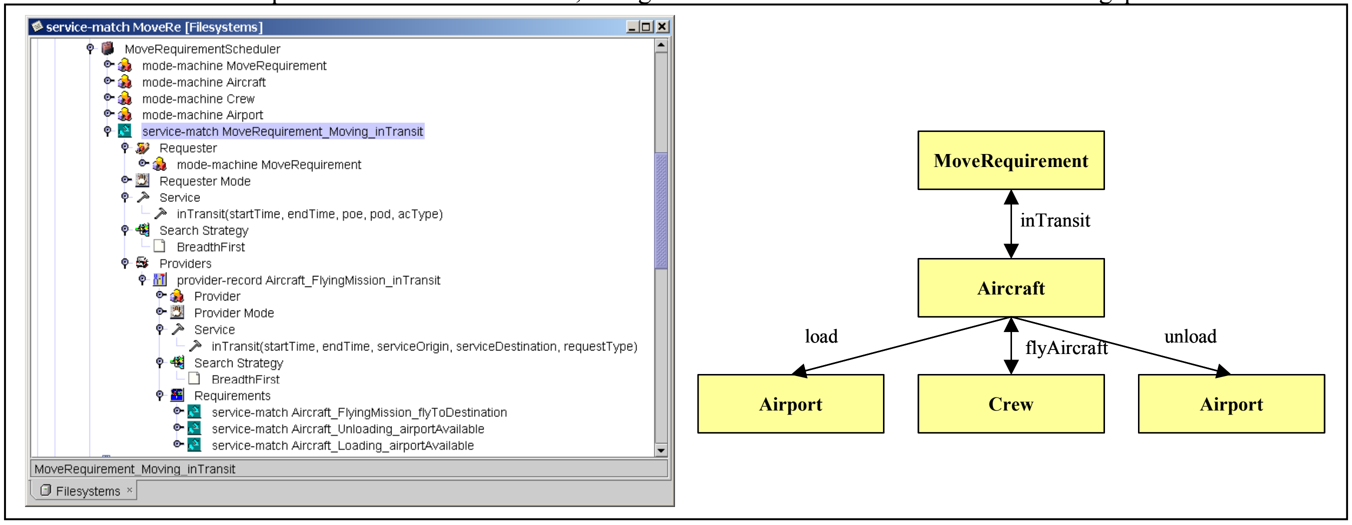
3. Code Generation

From the activity machine models described in the previous section, Planware automatically generates a fully operational scheduling application. The key component of the generated code is a search-based scheduling algorithm, and a constraint propagation mechanism. In addition to the search algorithm implementation, support code is also generated to represent the resources and activities, and to produce I/O for the application. In the next paragraphs we will explain in detail the code generation process, and the different components created by Planware.

Planware generates search-based scheduling algorithms implementing a *bidding process* as its main control cycle. In this process, resources requiring services post tasks, or requests for bids. Provider resources capable of performing the type of service specified in a task respond with their best bid according to their own internal strategy. The requesters then collect the bids, rank them according to the requester's objective function, select the best bid, and notify the selected bidders. Constraint propagation is triggered every time a bid is accepted. The propagation updates the internal state of the resources involved in the bidding. The rejected bids are discarded and no additional work is needed.

The concrete implementation of the scheduling algorithm used in a particular application is obtained by instantiating and composing *program schemas.* A program schema is a parameterized fragment of algorithmic logic that gets instantiated for each service match between a given provider and requester. Different program schemas are used to allow a number of different bidding generation and bidding selection mechanisms to be combined in the implementation of an application. For example, a program schema could be used to implement a bidding mechanism in which the first feasible bid is accepted; a different one could collect up to *n* bids, and select the one that can finish the service with the minimum amount of time; a third one could generate all possible bids and select the one with earliest start-time.

The program schemas are composed in a tree-like structure reflecting the structure of service matches defined by the activity machine models. As described in the previous section, an activity machine can request or provide services. For each required service in the model, the generator code will search for a matching provider — a



```
service-match MoveRe [Filesystems]
MoveRequirementScheduler
  mode-machine MoveRequirement
  mode-machine Aircraft
  mode-machine Crew
  mode-machine Airport
  service-match MoveRequirement_Moving_inTransit
    Requester
      mode-machine MoveRequirement
    Requester Mode
    Service
      inTransit(startTime, endTime, poe, pod, acType)
    Search Strategy
      BreadthFirst
    Providers
      provider-record Aircraft_FlyingMission_inTransit
        Provider
        Provider Mode
        Service
          inTransit(startTime, endTime, serviceOrigin, serviceDestination, requestType)
        Search Strategy
          BreadthFirst
        Requirements
          service-match Aircraft_FlyingMission_flyToDestination
          service-match Aircraft_Unloading_airportAvailable
          service-match Aircraft_Loading_airportAvailable
MoveRequirement_Moving_inTransit
Filesystems
```

Figure 3 Service Match Interface and Schematic Service Tree

resource providing a service that matches the signature of the required service. As a provider for a given service can request additional services from other resources, the service matches define a direct acyclic graph we refer to as the *service-match tree.* The code generator traverses the service tree creating the appropriate code to formulate the task, generate the bids, select and accept the bid.

The service-match tree is an auxiliary data-structure created by Planware before the actual application generation. By exposing the structure of the service-match tree to the user through the graphical interface, a greater level of control over the code generation can be obtained. An advanced user can configure the search schemas to be used by the generator for each service match in the tree. Figure 6 shows the service match generated for a model in which an Aircraft resource can provide transportation services to a MovementRequirement task, and requires additional services from Crew and Airport resources. Through the GUI, the user can change the search strategy, as well as the sequence in which the services are satisfied.

The constraint propagation code used to update the resources is not automatically synthesized. A standard implementation of an arc consistency algorithm that propagates temporal constraints on a simple temporal network is used. All schedulers generated share the same implementation.

The constraint propagation is responsible for maintaining consistent start times for all scheduled activities. Each activity has a time bound representing the earliest and latest time the activity can start executing. Each activity time bound defines a node in the constraint network. The scheduler adds temporal constraints (arcs) between time bounds (nodes) as the problem solving process evolves. If constraint violations are detected, scheduling decisions are retracted, and the search backtracks to the last decision point before the violation.

Activities and resources are closely related. Resources are represented by a capacity profile: A temporal sequence of activities representing the resource reservations performed by the scheduler. The profile represents a trace of the activity machine defined in the abstract model. The data structure used to represent the profile must be optimized for lookup and update. During the bid creation phase of the scheduling algorithm, the providers inspect their capacity profile searching for feasible intervals capable of feasibly performing the requested service. Once the requester accepts a bid, the selected provider updates its own profile to reflect the new reservation. Planware uses a binary tree implementation optimized for the particular type of resource.

The representation of the activities in the resource profile is generated from the set of variables defined by the activity machine model. All activities created for a given resource instance share the same set of constants. Activities are defined as a product type (record structure) with a field for each variable in the model. The code to access and set each one of these fields is automatically generated. Additional code to print and display individual activities, and activities sequences is also generated to facilitate debugging, testing, and schedule visualization. A number of different output formats are supported: plain text, XML, etc.

Activities are dynamically created at schedule computation time. Activity sequences are created by the bidding mechanism previously discussed. The generation of the bid creation mechanism is one of the most complicated components of Planware. It involves the generation of code capable of querying the resource profile for feasible intervals, expanding the sequence of activities the resource must execute, and enforcing the constraints imposed on the service by both the requester and by the provider resource.

The bid creation mechanism is implemented as a 3-step process:

a. **Identify feasible capacity intervals:** Based on some approximated values of activity duration and capacity requirements, the algorithm queries the resource profile for capacity intervals capable of accommodating at least the approximated values. This approximation is used to avoid activity expansion for intervals already known to be infeasible.

b. **Expand activity sequence:** This is the most expensive function of the algorithm. Generating the activity sequence requires simulating the behavior described by the activity machine until an activity capable of performing the desired service can be generated. Since this expansion corresponds to a fragment of the resource behavior that will be inserted in the profile to define the global behavior, the sequence must also maintain consistency between previous and next activities in the profile.

c. **Propagate constraints:** Once the activity sequence is created, constraint propagation is performed to adjust the time bounds of newly created activities to reflect the constraints imposed by existing resource reservations, plus constraints imposed by the service requester.

If these three steps generate a feasible activity sequence, a bid containing this sequence is sent to the tasker resource. If the bid is accepted, then the resource profile is updated to include the new sequence of activities.

After a bid has been accepted, and the resources appropriately updated, the search algorithm change its focus to schedule additional pending service requirements. Depending on the configuration provided by the user through the service match, the search proceeds by scheduling the requirements of the current bidder, or goes back to the level of the previous requester, and schedule its next task. The sequence of services scheduled is determined by the service match structure.

In terms of the global behavior of the application, the execution of the generated scheduler starts by reading a file describing all the top-level tasks, and all concrete resources available. The scheduling algorithm cycles through the top-level tasks, and expands the search following the structure defined by the service-match tree. If a top-level task cannot be satisfied using the available resources, it is marked "unschedulable" and discarded. Once there are no more pending tasks in the system, the scheduling algorithm finishes its execution and, if instructed to do so, outputs the schedule in text form, writes the schedule to an XML file, and/or displays the schedule on the GUI.

In addition to the generated code and the constraint propagation code, a small number of library components representing time, capacity, and other utilities are also composed with the algorithm implementation to generate the final executable.

4. Implementation

Planware's implementation can be divided into two main components: the graphical interface and the code generator.

Planware Interface

The interface component is written in Java and uses the open source configurable IDE platform NetBeans, which is an extensible integrated development environment designed to support multiple programming languages and formalisms. Additional capabilities are added to the NetBeans platform by writing modules using its customization API. A NetBeans module is just a JAR file (collection of compressed java class files) that can be "installed" in the platform. A module can implement a number of different capabilities like syntax sensitive source code edition, compilation, execution, and debugging among others. The Planware module provides:

1. An outline editor for editing activity machines based on a hierarchical representation of the models.
2. A graphical editor that allows the visualization and fast specification of activity machines.
3. A source code editor for more detailed specification of the models.
4. Visualization tools for inspecting the results of executing the generated code on test data.

NetBeans also provides as part of its standard module library components to manipulate and visualize the XML files that are generated by the scheduler as output.

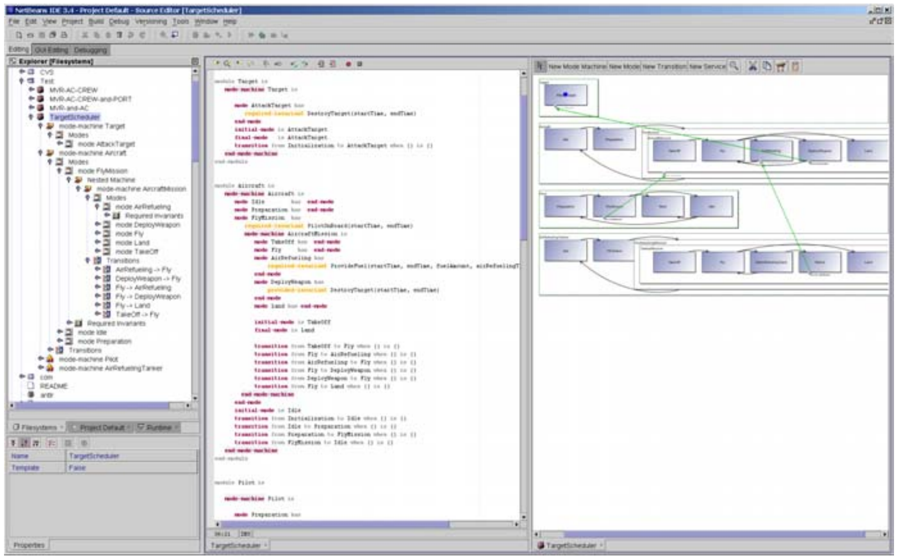
Developing resource models in this environment requires very little knowledge of Planware syntax. The set of syntax constructs is small and most of the model creation activity can be accomplished using just the outline and graphical editors.

A typical Planware resource model has less than a hundred lines of text, and can be created in a matter of minutes. A complete application model can be defined in few hours using a highly interactive environment.

The advantage of using an extensible platform like NetBeans is that the full application development and execution can take place in the same environment using the same interaction paradigm. Defining models, generating and compiling code, and executing the scheduler are all defined using the same basic set of actions and gestures.

Planware Code Generator

Planware code generator is implemented as an application layer on top of Specware, Kestrel's software synthesis platform [SJ96]. The code generator is a lisp process that communicates with the graphical interface through sockets. The interface sends commands to the lisp process to generate the scheduler code for a given model, to



```
module Target is
  mode-machine Target is

    mode AttackTarget has
      required-invariant DestroyTarget(startTime, endTime)
    end-mode
    initial-mode is AttackTarget
    final-mode   is AttackTarget
    transition from Initialization to AttackTarget when {} is {}
  end-mode-machine
end-module


module Aircraft is
  mode-machine Aircraft is
    mode Idle        has  end-mode
    mode Preparation has  end-mode
    mode FlyMission  has
      required-invariant PilotOnBoard(startTime, endTime)
      mode-machine AircraftMission is
        mode TakeOff has  end-mode
        mode Fly     has  end-mode
        mode AirRefueling has
          required-invariant ProvideFuel(startTime, endTime, fuelAmount, airRefuelingT
        end-mode
        mode DeployWeapon has
          provided-invariant DestroyTarget(startTime, endTime)
        end-mode
        mode Land has end-mode

        initial-mode is TakeOff
        final-mode is Land

        transition from TakeOff to Fly when {} is {}
        transition from Fly to AirRefueling when {} is {}
        transition from AirRefueling to Fly when {} is {}
        transition from Fly to DeployWeapon when {} is {}
        transition from DeployWeapon to Fly when {} is {}
        transition from Fly to Land when {} is {}
      end-mode-machine
    end-mode
    initial-mode is Idle
    transition from Initialization to Idle when {} is {}
    transition from Idle to Preparation when {} is {}
    transition from Preparation to FlyMission when {} is {}
    transition from FlyMission to Idle when {} is {}
  end-mode-machine
end-module


module Pilot is

  mode-machine Pilot is

    mode Preparation has
```

compile the generated code, and to execute the scheduler on some test data.

The Planware code generator translates the activity machines, and service match structure into an implementation of the algorithms and auxiliary data-structures described in Section 3. Planware first generates an intermediate representation of the algorithms in MetaSlang, the specification language used internally by Specware. This representation is then further refined, optimized, and composed with appropriate library code to generate a highly optimized implementation of the scheduling application in some programming language. Planware currently can only generate schedulers using CommonLisp as the target language. Current work at Kestrel is developing generators for C and Java.

For a problem model with 4 activity machines, specified by approximately 500 lines of text, Planware generates an intermediate representation with around 10,000 lines of code. The final CommonLisp code generated will be around 30,000 lines of code since all the library code used is included as part of the target implementation. The total synthesis time is a few minutes.

In terms of run-time performance of the generated schedulers, without any special heuristics added, models with four resource types running on data sets with thousands of tasks, and around 20 resource instances for each resource type, generate schedules in a matter of seconds. The run-time performance of the generated code was around 20% faster than the performance provided by scheduling applications previously developed manually by the authors for the domain of logistical deployment.

5. Concluding Remarks

From a synthesis point of view, Planware emphasizes the importance of a well-designed domain-specific requirement language. Planware provides an answer to the question of how to help automate the acquisition of requirements, and how to assemble a formal requirement specification. The key idea is to focus on a narrow, well-defined class of problems and programs, and to build a precise, abstract, domain-specific description formalism that covers this class.

One view of Planware is that it takes a model of a complex resource system, specified as interacting concurrent processes, and generates code that prescribes legal behaviors that respect constraints and service requirements. As such, Planware's modeling formalism supports scheduling, resource management, planning, and other domains, such as protocol design and code synthesis.

There are a number of directions that we are pursuing to further develop Planware. The schedulers generated by Planware are batch-oriented and they don't directly handle the re-scheduling that typically arises in practical applications. However, since the schedulers work by incrementally extending a partial schedule, it is possible to restart the scheduler with a modified schedule that reflects the current situation together with any new or modified tasks. The scheduler would then schedule forward as before, but taking the current situation into account. In current projects we are exploring this approach and other algorithms for handling incremental rescheduling problems. Issues of real-time and distributed scheduling are also being explored.

Another direction for future work is the extension of the current formalism with objective functions (minimum cost, minimum lateness, etc). The question is how to infer from the objective function a sorting criterion for bids in order to pursue a best-first search strategy.

In one aspect we have found the expressivity of the current activity machines a little limited. The activity machines in Planware are sequential in nature. However, there are certain classes of resources that are characterized by asynchronous concurrent reservations. Examples of such resources include fleets of vehicles, parking lots, and concurrently used power sources. This suggests extending the Planware activity machines to allow multithreading, and treating each reservation as a separate thread. Finally, other projects at Kestrel are developing generators of C and Java code from Specware's MetaSlang language. We hope to exploit this work in future version of Planware.

REFERENCES

[Blaine98] Lee Blaine, Limei Gilham, Junbo Liu, Douglas R. Smith, and Stephen Westfold, Planware --Domain-Specific Synthesis of High-performance Schedulers, *Proceedings of the Thirteenth Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, California, October, 1998, 270-280.

[Fischer00] B. Fischer, J. Schumann, and T. Pressburger, Generating Data Analysis Programs from Statistical Models, *Proc. ICFP Workshop on Semantics, Applications, and Implementation of Program Generation*, Springer LNCS, 2000.

[Lowry94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood: AMPHION: Automatic programming for scientific subroutine libraries. In Proc. 8th Intl. Symp. Methodologies for Intelligent Systems, Springer LNAI 869, 326–335, 1994.

[Pavlovic01] Dusko Pavlovic and Douglas R. Smith, Composition and Refinement of Behavioral Specifications, Proceedings of the Sixteenth International Conference on Automated Software Engineering, IEEE Computer Society Press, Coronado Island, CA, 2001, 157-165.

[SJ96] Srinivas, Y. V. and Jullig, R., "SPECWARE: Formal Support for Composing Software," in *Proceedings Of The Conference on Mathematics of Program Construction*, B. Moeller, Ed., LNCS 947, Springer-Verlag, Berlin, 1995, 399--422.

[Smith90] Smith, D.R., KIDS: A Semi-Automated Program Development System, IEEE Transactions on Software Engineering 16(9), Special Issue on Formal Methods, September 1990, 1024-1043.

[Smith96] 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.

[Smith97] Smith, S.F. and Becker, M.A., An Ontology for Constructing Scheduling Systems, in *Working Notes of 1997 AAAI Symposium on Ontological Engineering*, Stanford, CA, March, 1997 (AAAI Press).