Flexible, Constraint-Based Tools for Complex Scheduling Applications

Ora Lassila and Stephen F. Smith

Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract

In this paper, we advocate a view of scheduling application development as a differential and incremental process. We describe DITOPS, a flexible constraint-based transportation scheduling system that is designed to explicitly support this view, and has been used to provide a wide range of logistics support functionality.[1]

I. INTRODUCTION

During recent years, considerable progress has been made toward the development of more accurate and flexible knowledge-based tools for practical scheduling environments [15]. The advantage of heuristic scheduling procedures directly based on knowledge-rich models has been demonstrated in several domains. Development of techniques for incremental schedule revision [12, 14, 22] has provided scheduling functionality closely paralleling the inherently reactive nature of scheduling in complex domains.

Despite many promising results, widespread use of these scheduling techniques in operational settings has not occurred. The source of strength of knowledge-based scheduling tools (the incorporation of knowledge specific to the constraints and objectives of the target domain) also complicates system development. Though there are certainly degrees of commonality in the characteristics of different application domains, there is also variation. Diversity exists in the structure of different scheduling environments, in the constraints that dominate, in the important performance objectives, etc., all of which imply deviation with respect to decision support requirements, modeling assumptions, solution procedures and required knowledge and heuristics. Since different scheduling domains and application requirements invariably present unique challenges and complexities, efforts to apply knowledge-based scheduling technology operationally have remained time-consuming, *ad hoc* design and development projects.

In this paper we describe research aimed at addressing the difficulty of application development. We have developed a system called DITOPS, an interactive constraint-based tool for the generation, analysis and manipulation of military crisis-action transportation schedules. Though customized to specifically support user decision-making requirements and tasks in this domain, the DITOPS software technology base is at the same time explicitly designed for reconfigurability and reuse in other application domains. A general, constraint-based model of scheduling is used to provide an architectural structure for composing scheduling tools, which leads to a hierarchy of component protocols and the specification of functional components with varying degrees of inter-domain transferability. Through use of object-oriented implementation techniques, existing components (and protocols) are straightforwardly adaptable to match the specific characteristics of any given application domain. Before considering this software architecture in more detail, we first provide some perspective on our approach and summarize the capabilities of the DITOPS transportation scheduler.

II. RECONFIGURABILITY PROBLEM

Our approach to the rapid development of flexible scheduling systems derives from previous work with the OPIS scheduler [18]. OPIS implemented a reactive approach to scheduling, based on use of a set of solution procedures with differential optimization and conflict resolution capabilities. In OPIS, solution procedures are dynamically selected and applied to best respond to current (re)scheduling needs and opportunities. Scheduling (or rescheduling) proceeds iteratively, utilizing constraint management techniques to reflect the consequences of newly imposed (or changed) solution constraints

[1]The research reported in this paper has been supported in part by the Advanced Research Projects Agency under contract F30602-90-C-0119 and the CMU Robotics Institute. The authors are affiliated with the Center for Integrated Manufacturing Decision Systems of the CMU Robotics Institute. They can be reached through email at ora@cs.cmu.edu.

at each step and look-ahead analysis techniques to estimate critical tradeoffs and flexibilities. Comparative experimental studies conducted with OPIS in the context of specific manufacturing scheduling environments have verified the performance advantages of this constraint-based scheduling approach with respect to both predictive and reactive scheduling criteria [11, 14].

OPIS also defined a system architecture to support this opportunistic scheduling methodology, intended to enable the definition of customized application models and the introduction of application specific scheduling methods. From this standpoint, which relates directly to issues of reconfigurability, some design deficiencies can be retrospectively identified:

- The original design of the OPIS constraint modeling and management subsystem [9] traded off generality in the types of constraints that could be represented and managed, to achieve sufficient efficiency for the manipulation of full-scale production schedules. Although this functionality was quite satisfactory for most production scheduling applications, more recent work in adapting the scheduler to crisis-action deployment scheduling necessitated significant extensions to the constraint management infrastructure.
- As indicated, reactive scheduling was based on matching current rescheduling needs and opportunities to the differential capabilities of constituent revision procedures and heuristics. In providing a structure for specifying and coordinating reactive scheduling strategies, the supporting control architecture made several specific assumptions as to the mechanics of this process. Many aspects of this architecture reflect the original system design orientation toward providing incremental, reactive response to unexpected executional circumstances. Architectural commitments like this are much too strong, however, when viewed from a larger perspective of responding to ill-structured actions formulated by users.

The identified shortcomings highlight the need for stronger emphasis on configurability and extensibility in scheduling system design. Specializing component functionality (such as constraint propagation) is not a bad idea *per se* – it is often crucial to achieving efficiency – but the real problem lies in organizing system functionality so that component services appropriate to a given domain can be easily substituted and configured (for example, the work described in [10] achieves something to this effect). It is very difficult to anticipate all future needs for system extension; instead, a technique must be used that allows specialization, modification and extension of *any* component of the system. Within OPIS, *frame-based* representation techniques were used to provide both a repository of primitives for modeling domain constraints and a framework for specifying strategic control knowledge. These representational formalisms provided the structure to define flexible and expressive scheduling models, but no explicit mechanisms for encapsulation and information hiding. Pragmatically, this greatly complicates specification of, and adherence to, a layered model semantics, which is essential to the development of reconfigurable and extensible software systems.[2] As has been previously observed [3, 6, 7], modern *object-oriented* programming technologies provide a more direct and effective approach to specifying model semantics, through explicitly defined protocols for interaction with model components.

III. RECONFIGURABLE SCHEDULING SYSTEMS

The observed difficulties in the development of high-performance scheduling applications suggest that a considerable simplification of the application construction process is required. Despite the need for potentially highly specialized solution procedures and heuristics to achieve sufficient performance in any given decision-support context, our claim is that the solution structures required in various applications can be seen as more or less similar if they are viewed compositionally, and exploitation of this fact is the key to achieving broad applicability. It is possible to transform application construction into a differential and incremental process, emphasizing customization and reuse of component functionality. This leads to the notion of a *reconfigurable scheduling system*: An application development framework combining a "toolbox" of basic modeling and scheduling primitives with explicit protocols for assembling, aggregating and specializing these primitives to configure the decision support services [16]. The general philosophy of constructing applications is to build increasingly complex and specialized services, ultimately resulting in an application. New services, as they are composed, are encapsulated as additional tools and are available for subsequent reuse (an application designer may accumulate a library of specialized

[2] In particular, if all the slots of a frame are accessible, no encapsulation is achieved; this results in fragile implementations that are difficult to modify and maintain.

classes, making his task easier as the library grows).

Along these ideas we have constructed a *reconfigurable framework* for the development of scheduling applications, based on object-oriented programming techniques and software reuse. The framework allows the application construction process to be differential and incremental in nature, primarily focusing on the differences between existing software and the system being constructed. Object-oriented programming techniques potentially provide high reusability of software, but only if the system design project places special emphasis on the design of reusable *components* (e.g., [21]). The design of these components must be carried out with generality and extensibility in mind. The reconfigurable framework introduces a *general scheduling ontology* which serves as the starting point for a more detailed analysis of the target domain. The framework offers the scheduling system designer a class library of general scheduling concepts, such as *activities*, *resources*, *products* and *orders*. In addition to a classification of concepts, the library provides functionality for these concepts, effectively defining their operational semantics. Constructing a scheduler using this approach consists of the following:

- *Selecting* suitable classes from the library, matching features of the target system with those of the library.
- *Combining* the selected classes into more complex services, using both conceptual – multiple inheritance – and structural – aggregation and delegation – techniques.
- *Extending* the existing classes for domain-specific functionality when necessary, typically by specializing or overriding methods provided by the library.

The core library provides a general scheduling ontology. This ontology is specialized for specific domains. To give an example, we have built a transportation domain ontology for the construction of transportation-related applications. The general and the domain-specific ontologies are used to build organization-specific ontologies and actual scheduling applications. Our system focuses only on scheduling and planning, yet in the context of manufacturing organizations this approach is in many ways similar to that of the CIM-OSA -architecture [4] (c.f. its *Generic Level*, *Partial Models* and *Particular Models*).

IV. The DITOPS Transportation Scheduler

DITOPS is a flexible, interactive tool for the generation, analysis and manipulation of transportation schedules in the military crisis-action planning domain. Its software architecture uses a flexible and general model of scheduling as an iterative, constraint-directed process. The implementation of DITOPS uses the reconfigurable framework: through the use of the class library, it provides an extensible modeling and constraint management framework which enables straightforward incorporation of the dominant constraints of a given scheduling application, a basic set of constraint analysis primitives, a set of incremental scheduling methods that provide differential optimization and conflict resolution capabilities, and an explicit set of protocols for integrating their use in different decision-support contexts.

DITOPS has been configured to provide a wide range of functionality:

- Generation of high-quality "TPFDD" (Time-Phased Force Deployment Data) schedules – DITOPS is capable of generating large-scale deployment schedules that account for a wider range of constraints than do current simulation-based tools (improving the reliability of results), while producing better performance profiles (e.g., schedules with fewer late arrivals) and extending the range of decisions that are made (e.g., to include determination of travel mode – sea or air) [19].
- Reactive revision of deployment schedules – DITOPS is also capable of incrementally revising schedules in response to changed constraints. These capabilities provide a basis for mixed-initiative analysis and improvement of large scale schedules as well as efficient, coordinated response to unexpected events (e.g., port closings due to weather conditions).
- Resource capacity analysis – DITOPS has been used to analyze and detect resource capacity bottlenecks in conjunction with higher level Course of Action (COA) planning.
- Plan feasibility checking – DITOPS has also been configured to perform interactive COA employment plan feasibility checking and conflict diagnosis (the reader is referred to Appendix A for an example of how the reconfigurable framework was used to implement this).

In large-scale crisis-action planning (or in plant-wide manufacturing management, for that matter), it is unreasonable to expect planners to comprehend schedules – and meaningfully interact with a scheduling system – at the level of the system solution

model; there are far too many decisions and details, most of which are unimportant from the standpoint of user tasks and goals, and the complexity of decision-making at this level is overwhelming. Users must necessarily operate at higher, task-oriented levels, while decision-support tools must bridge the gap between user and system models of schedules and decision-making, and "manage the details" in accordance with user goals and intentions.

In DITOPS, interaction between a user and the system occurs through a *direct manipulation* interface which emphasizes visualization and manipulation of schedules in terms of resource capacity utilization over time. Based on a hierarchical resource model, the user can create resource capacity views at various levels of aggregation:

- The user can select temporal intervals by "boxing" the area of interest with the mouse. Any querying and manipulation of schedules and solution constraints is based on these time selections. For example, if the resource is an individual craft asset the transport activities supported by scheduled trips are accessible. At aggregate resource levels, graphical displays of various properties of the solution can be retrieved. This provides a basis for identification of solution deficiencies.
- User manipulation of schedules and problem constraints also centers around a selected resource profile interval. A resource can be made unavailable over a selected interval, causing any resulting inconsistencies in the schedule to be highlighted. Conversely, resource capacity of a given group of resources can be increased for a specified interval by moving to the appropriate aggregate resource display (this translates to adding craft to a fleet).
- Default rescheduling biases can be adjusted through a "slider" display which represents the relative importance to be attributed to each system known preference. In imposing any given change to the current schedule, there is no obligation to the user to provide additional revision constraints and guidance; in general, user decisions along these lines are considered to be defaults until they are changed.

The flexible environment provided by DITOPS closely matches the characteristics and requirements of decision making in complex scheduling domains. DITOPS handles the details of the user's higher-level actions by applying appropriate rescheduling procedures at each step to impose the changes specified by the user, and provides localized consequences of each change. Look-ahead analysis and scheduling techniques enable identification of principal causes of observed solution deficiencies, analysis of decision-making options and assessment of solution sensitivity to various events. More details about the decision support capabilities of DITOPS can be found in [17].

V. SOFTWARE ARCHITECTURE

Developers of scheduling systems are faced with decisions on two different levels of software architecture: (1) the internal architecture of the scheduling kernel and (2) the relationship of the scheduling kernel to other components of the system, especially in terms of interfaces (user interfaces, database interfaces, etc.). The internal software architecture of a typical scheduling system constructed using the reconfigurable framework has a layered structure, shown in Figure 1 (correspondingly, this is also the way the class library of the reconfigurable framework has been organized). From the bottom up, these layers are:

- An *Object System* – the lowest level consists of an object-oriented language and an object system (currently, we use CLOS, the Common Lisp Object System [20], augmented with features that make the construction of complex hierarchical models easier [8]).
- *Basic Services* are built using the object substrate. These are more like a general purpose class and function library and are not specific to scheduling.
- *Scheduling Class Library*, providing representation and modeling primitives, as well as classes and primitives for control. The representation classes consist of concepts like *resources*, *opera-*

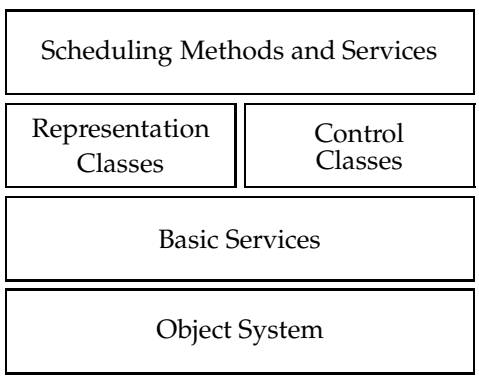

Figure 1: Kernel software architecture

tions, *orders* etc.
- Actual *Scheduling Methods and Services* are built using the lower-level layers. These services consist of different scheduling algorithms as well as support modules (such as bottleneck analyzers, for example).

A. Class Library Design

Different schools of thought exist as to how the object-oriented design process should be structured (e.g., [1, 21]). Mainly these differ just in details, the overall process consisting of two main steps: First, the proper *concepts* have to be identified, clearly specifying the structural and behavioral roles of each concept, and encapsulating them as *classes*. Second, a *protocol* of interaction between the classes has to be identified and specified. In the design of a class library, a protocol consists of a set of methods which are chosen to allow system extension. In other words, the protocol must be as general as possible, without compromising performance.

By specifying and documenting the protocol of interaction between different objects, the implementation of the class library (and thus the implementation of the scheduler) is "opened up" (allowing client programmers to extend it as they see fit), thus departing from the traditional "black box" approach to modular software. In other words, we allow limited visibility *inside* the library. This approach is similar to the *metaobject* implementation techniques of CLOS [5], where the objects of the implementation (also called *metaobjects*) allow the behavior of the system to be controlled and modified. If we think of traditional software reuse (e.g., [2]), this clearly goes beyond the simple function library approach.

The classes in our library have various roles and functions. A large number of classes is intended for modeling production systems, but will also function as providers of important services in a scheduling application. Some classes provide control functions, and serve as the building blocks of control architectures. Finally, instances of some classes are "pure" metaobjects: their role is to provide certain services (e.g. time and calendar services, error handling) and exist as vehicles for system extension (the client programmer can substitute them with objects from his own classes to provide modified or extended functionality). In general, most classes can be roughly divided into three main categories:

- *Base Classes* are (usually) not instantiable; they establish protocols and provide base and default functionality of a particular concept. They are easy to understand and use, since they each implement a limited, reasonably well defined set of functionality. Complex concepts have been broken into several classes.
- *Specialized Classes* can be instantiated, i.e. used without specialization. They typically refine and extend the protocols of the base classes. Many classes in the domain-specific ontologies belong to this category.
- *Mixin Classes* cannot be used alone, they must always be "mixed" with base and specialized classes. They typically implement functionality, completely or nearly orthogonal to the base classes, or provide simple refinements to the base behavior.

Currently our core library has over 200 classes available for the client programmer. The specific model of military transportation planning, which is used in our transportation scheduler, has 70 additional classes, all specializations of the core classes.

The reconfigurable framework and its class library establish a full hierarchy of protocols implementing a model of constraint-based scheduling. They provide a starting point for a scheduling application developer, who will replace abstract classes with more specialized classes that suit the problem at hand. This approach is analogous to modern application user-interface development frameworks (such as MacApp [13], for example), which provide the basic functionality of a complex user-interface in the form of an application "skeleton". In our case, the "skeleton" is an empty, generic constraint-based scheduling system. Solutions and classes defined in the framework will probably suit the majority of application needs. However, there is always the possibility to replace any component of the system with a different or more specialized component. This flexibility is directly attributable to the abstract layering of object interaction protocols provided by the framework.

To replace a certain component in the abstract framework, it is necessary to (1) decide which protocols the new component will commit to, and (2) define or specialize an appropriate new class. Considering the structure where primitive classes and services are combined into increasingly complex services, there are two choices in replacing a component:

- The component commits to both upper and lower level protocols; in this case only the component itself needs to be replaced. This is possible since the internal protocols of the frame-

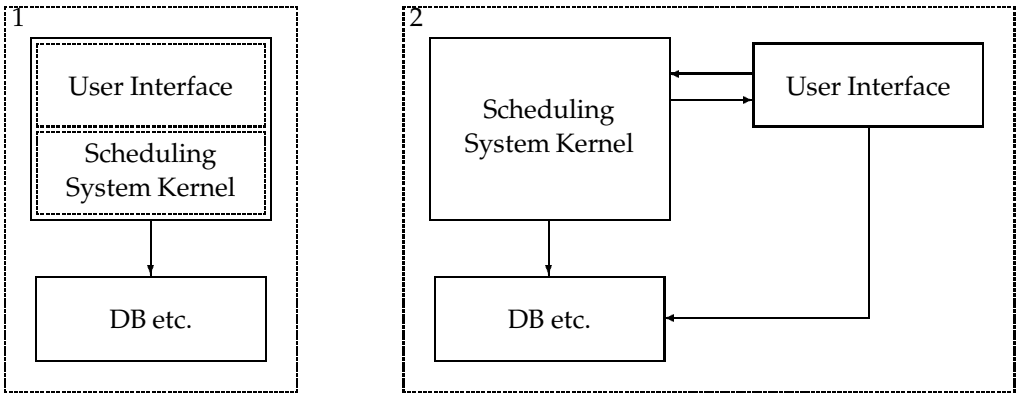

Figure 2: Alternate system architectures

work are documented.

- The component only commits to the upper level protocol; in this situation the lower levels of the service are either implemented from scratch or possibly with the help of some of the existing lower level services.

By providing a complete scheduling system framework we provide a structural backbone for scheduling system (re)configuration. Application development becomes a differential process, since all the basic services already exist, and only differences to the target scheduler need to be addressed. Appendix A gives concrete examples of the use of the class library.

B. Scheduler Interfaces

As observed with DITOPS, the need for a scheduling system to flexibly support the user's decision making process gives raise to the requirement of efficient user interaction. Currently, DITOPS implements the user interface as a layer on top of the core scheduling functionality. An alternate approach, which we have started to pursue, is to separate the user interface into its own process; this way, the scheduling kernel can be seen as a server. The "mixed-initiative" philosophy forces us to depart from a classical *client-server* architecture – a more symmetric relationship between the scheduler and the user interface is required.[3] Different approaches to system architectures are depicted in Figure 2.

As noted, interfaces to other system components are also needed in a full-fledged scheduling system and decision support tool: these interfaces, such as interfaces to databases, can be implemented using object-oriented techniques to allow a high degree of configurability.

[3]Obviously, "mixed initiative" implies that also the scheduler can initiate actions, and may thus require the services of the user interface without any initiative from the user.

VI. CONCLUSIONS

In this paper the difficulty of developing knowledge-based scheduling systems for decision support has been addressed, advocating the concept of a *reconfigurable scheduling system* as a means of simplifying the application development process in different scheduling domains. A constraint-based scheduling model as a basis for future scheduling systems provides a structure for reconfiguration. Modern object-oriented techniques enable reconfigurability and allow "design for reuse" – in the long run this makes it possible to rapidly deliver reliable decision-support applications.

Starting application development from a "skeletal" scheduling system and using object-oriented construction techniques – specialization of base classes for problem-specific functionality, for example – allows development to be transformed into a differential and incremental process. The components of the development system, however, have to designed with extension requirements in mind to achieve a sufficiently open implementation.

The modular implementation technique can also be taken a step higher by viewing different components of a complete information system (a production management system, for example) as reconfigurable components. The same principles for design (an open, extensible implementation with detailed specification of interaction protocols) as used for the basic software components can also be used here. Since the higher-level components are more complex in nature (and thus more specific) than individ-

ual object classes of the implementation, it can be expected that direct possibilities of reuse are fewer. With an open implementation technique, however, even highly specialized components can be reconfigured and thus possibly reused.

References

[1] Grady Booch, *Object-Oriented Design with Applications*, Redwood City (CA), Benjamin/Cummings, 1991.

[2] M.F. Bott and P.J.L.Wallis, "Ada and Software Reuse", in *Software Reuse with ADA*, (eds. R.J. Gautier and P.J.L. Wallis), IEE, 1990.

[3] Juha Hynynen and Ora Lassila, "On the Use of Object-Oriented Paradigm in a Distributed Problem Solver", *AI Communications* 2(3) pp. 142-151, 1989.

[4] H.R. Jorysz and F. Vernadat, "Defining CIM Enterprise Requirements using CIM-OSA", in *Computer Applications in Production and Engineering: Integration Aspects (CAPE'91)*, (eds. G. Doumeingts, J. Browne and M. Tomljanovich), Elsevier Science Publishers, 1991.

[5] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge (Mass.), 1991.

[6] Ora Lassila, "Frames or Objects, or Both?", in *Workshop Notes from the 8th National Conference on Artificial Intelligence (AAAI-90): Object-Oriented Programming in AI*. Boston (MA), July, 1990. [Report HTKK-TKO-B67, Department of Computer Science, Helsinki University of Technology, Otaniemi (Finland)].

[7] Ora Lassila, "The Design and Implementation of a Frame System", Faculty of Technical Physics, Helsinki University of Technology, Otaniemi (Finland), Master's Thesis, 1992.

[8] Ora Lassila, *Parsifal Object System – Programmer's Reference Manual*, CMU Robotics Institute Technical Report, in preparation, 1994.

[9] Claude LePape and Stephen F. Smith, "Management of Temporal Constraints for Factory Scheduling", in *Proceedings IFIP TC 8/WG 8.1 Working Conference on on Temporal Aspects of Information Systems (TAIS 87)*, 1987.

[10] Claude Le Pape, "Using Object-Oriented Constraint Programming Tools to Implement Flexible 'Easy to Use' Scheduling Systems", in *Proceedings NSF Workshop on Intelligent Dynamic Scheduling for Manufacturing Systems*, Cocoa Beach (FL), 1993.

[11] P.S. Ow and S.F. Smith, "Viewing Scheduling as an Opportunistic Problem Solving Process", in *Annals of Operation Research* Vol. 12, (ed.) R. Jareslow, Baltzer Scientific Publishing Co., 1988.

[12] Norman Sadeh, "Micro-Opportunistic Scheduling: The Micro-BOSS Factory Scheduler", in *Intelligent Scheduling*, (eds. M. Fox and M. Zweben), Morgan Kaufmann Publishers, 1993.

[13] Kurt. J Schmucker, *Object-Oriented Programming for the Macintosh*, Hasbrock Heights (NJ), Hayden Publishing Company, 1986.

[14] Stephen F. Smith, P.S. Ow, N. Muscettola, J.Y. Potvin, and D. Matthys, "An Integrated Framework for Generating and Revising Factory Schedules", *Journal of the Operational Research Society*, 41(6), 1990.

[15] Stephen F. Smith, "Knowledge-Based Production Management: Approaches, Results, and Prospects", *Production Planning and Control*, 3(4), pp. 350-380, 1992.

[16] Stephen F. Smith and Ora Lassila, "Configurable Systems for Reactive Production Management", in *Knowledge-Based Reactive Scheduling*, IFIP Transactions B-15, Amsterdam (The Netherlands), Elsevier Science Publishers, 1994.

[17] Stephen F. Smith and Ora Lassila, "Toward the Development of Flexible Mixed-Initiative Scheduling Tools", in *Proceedings of the ARPA/Rome Labs Planning Workshop '94*, Tucson (AZ), February, 1994.

[18] Smith, S.F., "OPIS: A Methodology and Architecture for Reactive Scheduling", in *Intelligent Scheduling*, (eds. M. Fox and M. Zweben), Morgan Kaufmann Publishers, 1993.

[19] Stephen F. Smith and Katia P. Sycara, *A Constraint-Based Framework for Multi-Level Transportation Scheduling*, CMU Robotics Institute Technical Report, 1993.

[20] Guy L. Steele, Jr., *Common Lisp – the Language* (second edition), Bedford (MA), Digital Press, 1990.

[21] Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs (NJ), 1990.

[22] Monte Zweben, Eugene Davis and Michael Deale, *Iterative Repair for Scheduling and Rescheduling*, Technical Report, NASA Ames Research Center, Moffett Field (CA), 1991.

A Examples of Class Library Usage

Use of the scheduling framework is best demonstrated through examples, demonstrating the "differential" construction process: The library provides concepts and mechanisms common to scheduling applications; to construct a specific application only differing concepts and behavior have to be defined and implemented. Reusability of the library components enables us to construct scheduling applications rapidly and reliably.

As the first example we describe a module implemented for a Course of Action planning application. Its purpose is to perform feasibility checks on COA employment plans generated by human planners. Both temporal feasibility as well as resource availability are verified. This system roughly consists of three parts:

- An *input module*, responsible for parsing the input from the planning application and instantiating a model of the plan. Plan instantiation is a three-step process: (1) appropriate resource objects are created, (2) activities are created (naming the resources that support these activities) and (3) the activities are linked using temporal relations. All objects are created either from the core library classes directly or from classes derived from the core classes.
- A *checker module*, responsible for checking the feasibility of the plan. This module employs the *Time Bound Propagator*, one of the services of the core library, to detect any cycles in the activity graph and to establish temporal bounds for each activity. The propagator kernel is a constraint path-consistency algorithm for constraint graphs consisting of activities and temporal relations.
- An *output module*, responsible for gathering the results of the feasibility check and communicating them to the planning application. It collects the *conflicts* created by the propagator, and filters these to provide a description of the potential infeasibilities. It translates both the conflicts and the time bounds of all activities into an format understood by the client application.

Activities in the plan have both a maximum and a minimum duration, both of which are not necessarily known. To support this we define a specific activity class, based on one base class (a base class for activities) and one mixin class (which allows the storage and use of fixed lower and upper bound durations). The resulting class `plan-activity` (shown in Figure 3) behaves like the basic operation class, but is capable of storing (and interpreting) both a minimum and a maximum duration. Here the specialization is achieved by selecting appropriate base classes and combining them through multiple inheritance into a problem-specific class. More complex specializations are possible by extending the behavior of the objects (by overriding or specializing methods of the objects' protocols).

```
(defclass plan-activity
          (operation min-max-duration-mixin)
  ()                    ; no new slots
  (:default-initargs
    :setup-duration 0)) ; no setups needed
```

Figure 3: Definition of the `plan-activity` class

As another example, we consider a simple transportation model. Typically movement activities have a duration that depends on the distance covered and the speed of the transportation resource. The example implements this by overriding the default duration methods of the `operation` class (source code definitions for this simple transportation model are shown in Figure 4). The class library provides a base class, `location`, which can be used for objects of the *origin* and *destination*. The protocol also specifies how distances between two locations is computed, allowing for specialization in different location classes (for example, locations of seaports, cities, etc.) This somewhat simplified example provides an insight into how specialization can take place.

```
(defclass movement-activity (operation)
  ((origin
     :initarg :origin
     :reader operation-origin)
   (destination
     :initarg :destination
     :reader operation-destination)))

(defclass transportation-resource
          (batch-resource)
  ((velocity  ; in distance units per hour
     :initarg :velocity
     :reader resource-velocity)))

(defmethod operation-compute-run-duration
           ((op movement-activity)
            (res transportation-resource))
  (* (/ (location-distance-between
          (operation-origin op)
          (operation-destination op))
        (resource-velocity res))
     (/ (time-units-per-day) 24)))
```

Figure 4: Definition of a "transportation" model