Configurable Systems for Reactive Production Management

Stephen F. Smith and Ora Lassila

Center for Integrated Manufacturing Decision Systems
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA

In this paper, we outline current work into the development of *reconfigurable scheduling systems*: systems that enable rapid customization to specific production environments through encapsulation, extension, and reuse of component scheduling "services" (e.g., domain modeling primitives, constraint management and analysis techniques, solution subprocedures, problem decomposition and configuration heuristics). Our approach is grounded on two basic premises: (1) that system organization and decision-support "services" should directly reflect the inherently reactive nature of decision-making in practical scheduling environments, and (2) that diversity in the character and requirements of different application environments will invariably require different specialized scheduling support services. We adopt a general constraint-based model of scheduling as an iterative, opportunistic process of schedule revision, which provides an architectural framework for formulating the configuration problem, and utilize object programming techniques to compositionally construct component services. Our overall goal is the development of an application building environment, which combines a "tool box" of basic modeling and scheduling primitives with facilities for assembling, aggregating and specializing these primitives to define the decision support functionality (or services) required in a given application context. As new services are composed, they can be encapsulated as additional, higher level tools and are available for reuse in subsequent applications.

Keyword Codes: D.1.5; I.2.1
Keywords: Programming Techniques, Object-oriented Programming; Artificial Intelligence, Applications and Expert Systems

1. INTRODUCTION

Two broad observations can be made with respect to the scheduling problems that must be faced in practical production environments:

- They can rarely be treated strictly as optimization problems. To be sure, optimizing capabilities are important, but the overarching goal of producing a good manufacturing system *behavior* requires larger attention to the ongoing scheduling process.

Schedule construction in practice tends to be an dynamic reactive process, involving negotiation among decision-makers with different organizational responsibilities and goals (e.g., marketing, engineering, production). An initial schedule is built, problematic or unsatisfactory aspects of the result are identified, requirements are relaxed or strengthened, schedule modifications are made and so on. In this context, the current schedule serves as an important nominal reference for identifying and specifying changes. While the focus is on improving the acceptability/quality of the solution, considerable pragmatic value is placed in retaining continuity in the solutions produced across iterations. Likewise, as unexpected events occur in the execution environment, it is important to preserve continuity in domain activity while making those schedule changes necessary to insure continued feasibility and attendance to overall performance objectives. Both of these aspects of the scheduling process place a premium on incremental, reactive scheduling capabilities.

- There is considerable diversity in the character of the scheduling problems faced in different manufacturing environments. While commonality can be found in overall solution and decision-support requirements, the structure of the manufacturing system, the types of constraints that dominate the problem, the types of uncertainties that must be confronted and the desired scheduling objectives vary considerably from one environment to the next. The utility of a given scheduling technology will directly depend on the level of fidelity of its underlying model with the operating constraints and conditions of the target manufacturing environment. Accordingly, we can expect that the heuristics and solution procedures required for effective decision-support will likewise vary from application to application.

The requirements implied by these observations are at odds with the decision-support capabilities provided by most contemporary production management tools. Existing systems generally operate with respect to simplified models of domain constraints, limiting their relevance and utility to operational decision-making. Moreover, existing systems are more often than not based on the use of inflexible schedule generation procedures (e.g., back-end optimizers or simulators). Such approaches force the user into an indirect and inefficient reactive decision-making cycle (where the user must hypothesize parameter changes that will bias the procedure toward the desired solution, run the procedure, interpret the results and iterate), and such approaches provide no support for maintaining stability in the solutions produced over time.

Recent research in knowledge-based scheduling has made progress toward overcoming these inadequacies [22]. The advantage of heuristic scheduling procedures that are directly based on knowledge-rich models of the actual operating constraints and objectives of the target environment has been demonstrated in several application domains. Similarly, recent development of techniques and heuristics for incremental schedule revision (e.g., [3, 4, 16–18, 20, 21, 26] has provided scheduling functionality that more closely parallels the inherently reactive nature of the scheduling process.

Despite the promise of these results, their impact in operational production management settings remains low. Somewhat ironically, the source of strength of knowledge-based reac-

tive scheduling techniques in overcoming the inadequacies of current production management technologies - incorporation of knowledge about the constraints and objectives of the particular manufacturing environment - also complicates the application building process. As observed above, different production environments invariably present different scheduling challenges and complexities, and efforts to apply knowledge-based reactive scheduling technology to specific operational environments largely remain time-consuming, "one-off" design and development projects.

In this paper, we describe work aimed at simplifying this application building process. Starting from a flexible and general model of reactive scheduling as an iterative, constraint-directed schedule manipulation process, we focus on the development of a system architecture that promotes appropriate instantiation of this model in specific manufacturing environments. In contrast to the perspective of a universally applicable heuristic search framework that can be programmed with the evaluation criteria of a given application (e.g., [6, 26]), our approach presumes that different applications will typically require different (more specialized) search and solution procedures, and it focuses on providing support for this solution configuration process. Our overall goal is to develop a *reconfigurable scheduling system*, an application building environment which combines a "tool box" of basic modeling and scheduling primitives with facilities for assembling, aggregating and specializing these primitives to configure the decision support functionality (or services) required in a given application context. As new services are composed, they can be encapsulated as additional, higher level tools and are available for reuse in subsequent applications.

2. CONSTRAINT-BASED SCHEDULING MODELS

In arguing against universal solution procedures, we are not suggesting that there is no commonality in the scheduling functionality required in different application domains. In fact, we believe quite the opposite; that required solution procedures 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. To do so, however, first requires that one settle on an appropriate scheduling model.

Constraint-based frameworks provide a model well-suited to the reactive decision-making requirements of practical scheduling domains. In broadest generality, this model defines a problem solving organization that distinguishes two components: a *decision-making* component, responsible for making choices among alternative scheduling decisions and retracting those that have since proved undesirable, and a *constraint management* component, whose role is to propagate the consequences of decisions and incrementally maintain a representation of the current set of feasible solutions (detecting inconsistent solution states when they arise). Schedule construction, revision, and improvement proceed iteratively within a basic *decide and commit* cycle. Most project management tools and several interactive scheduling systems [7, 15] are direct implementations of this model, with the user as the decision-making component.

Of more interest and importance in most scheduling environments, are extensions of this

basic model that off-load more decision-making responsibility to the system. The "spreadsheet" style of interaction provides a natural framework for "what-if" experimentation and iterative solution development, but interaction and decision-support is typically required at a much more aggregate decision-making granularity. One extension to the basic constraint-based model that preserves the "direct manipulation" style of interaction is one that refines the decide step of the cycle into a two-step process of formulating and executing decision-making actions. Within this model, *action formulation* is concerned with isolating a particular subproblem and *action execution* results in solution of this subproblem. The user, who interacts with system processes by formulating actions, is able to manipulate solutions in terms of higher-level (and more comprehensible) task-oriented perspectives (e.g., reschedule job x to start tomorrow), with the system's subproblem solution procedures providing, from the user's viewpoint, an amplification of deductive constraint management functionality; a more sophisticated (and typically heuristic) "propagation of effects".

In the simplest case, there is a direct mapping between user specifiable actions and system solution procedures, in which case the user holds complete responsibility for action formulation. The COMPASS interactive scheduling framework [5], for example, is organized in this fashion. In our view, however, the user should be able to operate in terms of much more ill-structured action specifications (e.g., relax weekend capacity constraints and reschedule late orders while minimizing overtime personnel, resolve the conflicts introduced into the schedule by the breakdown of machine x). This, more flexible viewpoint implies that the system must participate actively in structuring the appropriate subproblem to solve (e.g., in determining the appropriate scope of change, in translating objectives and preferences into appropriate heuristic revision procedures), and that subproblem solution may require coordinated execution of several solution procedures.

3. THE RECONFIGURABILITY PROBLEM

Our perspective on the development of reconfigurable scheduling systems derives from previous work with the OPIS scheduler [20, 23], which advocated a similar (but narrower) perspective toward reactive scheduling system development. OPIS implements a framework for incremental, reactive scheduling based on the use of a set of solution subprocedures with differential optimization and conflict resolution capabilities, which are dynamically selected from and applied to best respond to current (re)scheduling needs and opportunities. While the principal emphasis in this research was on mechanisms and heuristics for integrating methods to balance scheduling, schedule stability and system responsiveness objectives, this opportunistic, multi-perspective scheduling methodology, as well as the supporting architecture developed for specification and coordination of system activity, are directly relevant to the larger problem of configuring scheduling services for various application domains.

At the same time, we can retrospectively identify several design deficiencies in the OPIS scheduler from the standpoint of reconfigurable scheduling systems:

- Inflexibility in solution representation/constraint management assumptions - In the original design of the OPIS constraint management subsystem [13], specific assumptions as to solution representation were adopted and specialized constraint propagation and conflict detection mechanisms were developed to manage these representations in response to solution changes; in essence, generality in the types of constraints that could be represented and managed was traded off to achieve sufficient propagation efficiency to accommodate manipulation of full-scale production schedules. This functionality has proven quite satisfactory across a number of distinct production scheduling application domains. However, more recent work in adapting the original manufacturing production scheduler to the domain of crisis-action deployment scheduling (admittedly a quite different problem domain with quite different solution management requirements), necessitated fairly significant extensions to this constraint management infra-structure, which in turn resulted in a rippling of changes to other dependent system components (e.g., conflict analysis routines). Interestingly, the system's original solution subprocedures proved quite effective once these infra-structure adaptations were made, yielding better solutions than existing transportation scheduling tools in a comparative performance study [24]. The point here is not that specializing kernel system functionality such as constraint propagation is a bad idea; specialization is often crucial to achieving sufficient system efficiency. The problem lies in appropriately organizing system functionality so that component services (e.g., solution representations, constraint management techniques) appropriate to the domain under consideration can be easily substituted and configured. In this regard, the recent work of LePape is a strong step in the right direction [14].
- Overcommitment in the control infra-structure - Reactive scheduling in OPIS is based on a fairly general process of matching current rescheduling needs and opportunities (as inferred through analysis of the characteristics of the current solution state) to the differential capabilities of constituent revision subprocedures and heuristics. However, in providing a structure for specifying and coordinating reactive scheduling strategies, the supporting OPIS control architecture makes several specific assumptions as to the mechanics of this process (e.g., how external events are mapped to solution structures, how problems are detected, how the focus and scope of revision tasks are determined, how the ripple effects of local revision actions are accounted for). Many aspects of this architecture reflect the original system design orientation toward systems for incremental, reactive response to unexpected executional circumstances. However, when viewed alternatively from the larger perspective of supporting response to ill-structured user formulated actions, these architectural commitments are too strong. In fact, as was argued above with respect to underlying solution representations and constraint management techniques, we expect that user requirements in different application environments will imply different levels of architectural commitment. Reconfigurability requires a system design that retains the flexibility to adapt the mechanics of the control cycle to best fit the user requirements of specific domains.

Both of these deficiencies point to one overarching problem: the system design does not

properly anticipate the potential necessity for change/extensibility at any level of system functionality. Configuration flexibility is only provided with respect to pre-identified functional components (e.g., the addition/substitution of alternative solution subprocedures and alternative parameterizations of these subprocedures, specialization of domain descriptions to encode idiosyncratic application constraints), and the control infra-structure, while clearly relevant to an iterative, opportunistic model of user-system interaction, is too confining in its assumptions.

4. TOWARDS RECONFIGURABLE SYSTEMS

The shortcomings of the OPIS scheduler identified in the previous section highlight the need for stronger emphasis on configurability and extensibility in scheduling system design. 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 (i.e. method configuration and parameterization) control knowledge. While such representational formalisms provide the structure to define flexible and expressive scheduling models, they provide 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[1]. As has been previously observed [8, 11, 12], modern *object-oriented* programming technologies provide a much more direct and effective approach to specifying model semantics, through explicitly defined protocols for interaction with model components.

Our approach to simplifying knowledge-based scheduler construction relies heavily on object-oriented programming techniques and software reuse, allowing schedulers and other production planning applications to be constructed as a "differential" process, focusing primarily on the differences between existing software and the system being constructed. Object-oriented programming techniques can provide high reusability of software, but only if the system design project places special emphasis on the design of reusable *components* (e.g., [25]). The design of these components must be carried out with generality and extensibility in mind.

A scheduler construction project (potentially as part of a larger production management system design) begins with:

- Information gathering and *knowledge acquisition*, aiming to provide the system designer with a sufficient amount of detailed information about the production system.
- *Modeling* and *analysis* of the production system (e.g. production plant and process, logistics system).

[1]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.

In an object-oriented approach to software construction, these steps constitute an object-oriented analysis of the scheduling system (they also constitute part of the design). The approach taken in our reconfigurable framework is the introduction of a common scheduling and production planning ontology which serves as the starting point for a more detailed analysis of the target system. The system offers the scheduling system designer a class library of common and general scheduling concepts, such as *activities*, *resources*, *products* and *orders*. Constructing a scheduler or some other production planning application using our 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 (i.e. multiple inheritance) and structural (i.e. aggregation and delegation) techniques.
- *Extending* the existing classes to provide domain-specific functionality when necessary. Typically this is done by specializing or overriding methods provided by the library.

The basic class library provides a general scheduling ontology. This ontology can be specialized for specific fields of production (for example, we have built a transportation domain ontology, allowing us to easily build transportation-related applications). The general and the field-specific ontologies can then be used to build company-specific ontologies and actual production planning applications. It can be observed that although our system focuses on production planning and scheduling, this approach is in many ways similar to that of the CIM-OSA -architecture [9] (c.f. its Generic Level, Partial Models and Particular Models).

The general philosophy of application construction is to use the techniques described above to build increasingly complex (and specialized) services, ultimately resulting in an application. The classes designed and specialized in this process can be reused in subsequent applications. An application designer may accumulate a library of specialized classes, making his task easier as the library grows.

4.1. Library Design

Different schools of thought exist as to how the object-oriented design process should be structured (e.g., [1, 25]). 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. Second, a *protocol* of interaction between different 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" mentality to modular software, and allowing limited visibility inside different modules. This approach is similar to the *metaobject protocol* implementation techniques of CLOS [10], 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 goes beyond the simple function library approach, thus complicating liability issues, for example. These issues, however, are beyond the scope of this paper.

The classes in our library have various roles and functions. A large number of the 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 define 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. Examples of base classes are `resource`, `operation` and `conflict`.

- *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 field-specific ontologies belong to this category. Examples of specialized classes: `batch-resource`, `airport`, `resource-unavailability-conflict`.

- *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. Examples of mixin classes are `movable-resource-mixin` (to provide location tracking) and `min-max-duration-mixin` (to allow storage of lower and upper bounds of duration).

Currently our core library has 80 classes available for the client programmer. We have implemented a specific model for military transportation planning which has 40 classes, all specializations of the core classes.

4.2. Examples

The use and nature of the class library is best demonstrated through examples of how to select, combine and specialize classes and how to build services and applications. It is imperative to understand that building systems using a library like this is a "differential" process: The library provides concepts and mechanisms common to production planning

applications, and to construct an application only differing concepts and behavior have to be defined and implemented. Through reusability of the library components we are able to construct applications rapidly and reliably.

As an example of an application, we will discuss a module implemented for a larger planning application. The purpose of the module is to perform feasibility checks on plans generated by human planners using the planning application. 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. Three different things take place when a plan is instantiated: appropriate resource objects are created, activities are created (naming the resources that execute these activities) and the activities are linked using temporal relations. All objects are created either from the core library classes directly or from classes that have been specialized from the class library.
- A *checker module*, responsible for checking the feasibility of the plan. This module employs the *Time Bound Propagator*, one of the services provided by the class library, to accomplish two things: to detect any cycles in the activity graph, and to establish temporal bounds for each activity. The kernel of the propagator is a constraint path-consistency algorithm capable of propagating temporal constraints in a constraint graph 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 then translates both the conflicts and the time bounds of all activities into an appropriate format.

Activities in the plan have both a maximum and a minimum duration, both of which are not necessarily known. The next example shows how we define our own activity class, based on the base class `operation` and the mixin class `min-max-duration-mixin`:

```
(defclass plan-activity (operation min-max-duration-mixin)
  ()
  (:default-initargs
    :setup-duration 0))
```

The resulting class `plan-activity` behaves like the basic operation class, but is capable of storing (and interpreting) both a minimum and a maximum duration. Here the specialization has been effected by selecting the appropriate base classes and combining them through multiple inheritance to provide a problem-specific class. More complex specialization can be achieved by extending the behavior of the objects. This is done by overriding or specializing methods of the objects' protocols.

Let us consider scheduling material movement in the manufacturing domain. Movement activities typically have a duration that depends on the distance covered and the speed of the AGV resource. This can be implemented by overriding the default duration methods of the `operation` class. To illustrate this, we will give the definitions for a simple transportation domain:

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

(defclass agv (batch-resource)
  ((velocity              ; expressed in miles per hour
     :initarg :velocity
     :reader resource-velocity)))

(defmethod operation-compute-run-duration ((op movement-by-agv)
                                           (res agv)
                                           &optional type)
  (declare (ignore type))
  (* (/ (location-distance-between (operation-origin op)
                                   (operation-destination op))
        (resource-velocity res))
     (/ (time-units-per-day) 24)))
```

The class library provides a base class, `location`, which can be used for objects of the *origin* and *destination*. The protocol also specifies a method for locations, called `location-distance-between`, this can be specialized in different location classes (for example, locations on the factory floor, seaports, cities, etc.)[2]. This example is no doubt somewhat simplified but provides an insight into how specialization can take place.

Our last example should illustrate the mechanics of a lower-level protocol: A schedule is stored in *resource* objects by maintaining a representation of the available capacity of each resource over time. A protocol exists for querying this representation, allowing other program modules to find out about available capacity and windows of time where scheduling can take place (for example, the Time Bound Propagator relies heavily on this protocol). Each *operation* class has a method which queries the time available for the particular operation to be scheduled. Since operations know about their required resources, the method will call a resource method for querying time blocks. This method has several definitions, depending on the type of resource (for example, an "atomic" resource only allows one

[2]The function `time-units-per-day` used in the example is part of the time and calendar services of the library.

operation to be performed on it at a time, a "batch" resource allows time-synchronized execution of several operations, an "aggregate" resource represents a collection of resources, each capable of independently executing operations; in each case the available capacity is treated differently). A callback to the client is provided through the use of *time block strategies* which are objects that ultimately decide about time blocks found. The resource time block method provides this by calling `strategy-enough-capacity-p` and `strategy-check-blocks`. These methods allow, for example, capacity constraints to be relaxed, in order to resolve overconstrained situations.

To allow a scheduler to hypothesize about, and schedule into, infinite capacity, the predicate method `strategy-enough-capacity-p` may be overridden. This method determines whether the available capacity is large enough to allow allocation of the required capacity. In order to create the "illusion" of infinite available capacity, we may redefine this method to return true even in cases when there is not enough capacity available. To instruct the use of time intervals during which minimal overallocation takes place, the method `strategy-check-blocks` may be redefined: its role is to inspect time blocks and decide which ones are preferred.

The last example can be extended to implement a reactive scheduling technique where formerly allocated operations can be "bumped", i.e. the capacity they are reserving can be regarded free. To implement this, we need to assign priorities to each operation (or corresponding production plan), define a new strategy (say, `bumping-time-block-strategy`) and specialize the method `strategy-enough-capacity-p`. This method would, in case of inadequate amount of available capacity, find operations of lower priority whose capacity could be considered available for allocation.

4.3. Development Framework

The classes and protocols described in the previous examples are part of a larger framework, the *configurable scheduling framework*, which establishes a full hierarchy of protocols implementing the model of constraint-based scheduling developed in earlier sections. The framework provides a starting point for a scheduling application builder, who will replace abstract classes of the framework with more specialized classes that suit the problem at hand. This approach is analogous to modern micro-computer application development frameworks, which provide the basic functionality of a complex user-interface in the form of an application "skeleton" (e.g., [19]). 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 framework 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 building becomes a differential process, since all the basic services already exist, and only differences to the target scheduler need to be addressed. Of course, decisions also need to be made concerning interfaces to other systems (e.g. database interfaces, user interaction). We plan to extend the framework to provide predefined modules for these purposes in the future.

5. CONCLUSIONS

In this paper, we have advocated the development of reconfigurable scheduling systems as a means of simplifying the application building process in different production environments. To provide a structure for reconfiguration, we have adopted the constraint-based scheduling model as a likely candidate for the architecture of future scheduling systems. As pointed out, constraint-based techniques support the development of intelligent, localized rescheduling capabilities, thus providing an intuitive base for direct-manipulation, "spreadsheet-like" user interaction. Likewise, the basic division of problem solving architecture into decision making and constraint management components is compatible with the view of a scheduler as a decision support system rather than a fully automatic component of a production management system.

Reconfigurability can be seen as a means to rapid development of reliable scheduling systems and other intelligent production planning applications, and can also be viewed as a source of flexibility from the end user standpoint. The object-oriented techniques outlined in the previous section will serve as an enabling technology for reconfigurability, by transforming application development into a differential and incremental process, promoting software reuse, and ensuring unlimited extensibility.

REFERENCES

1. Booch, Grady, "Object-Oriented Design with Applications", Benjamin/Cummings, Redwood City, CA, 1991.
2. Bott, M.F. 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. Burke, P., and P. Prosser, "A Distributed Asynchronous System for Predictive and Reactive Scheduling", Tech. Report AISL42, Dept. of Computer Science, University of Strathclyde, 1989.

4. Collinot, Anne and Claude Le Pape, "Adapting the Behavior of a Job-Shop Scheduling System", *International Journal for Decision Support Systems*, 7(3), pp 341-353, 1991.
5. "Compass 2.0 User's Guide", Planning and Scheduling Group, McDonnell Douglas Space Systems Co., December 1992.
6. Fox, Mark S. and Katia P. Sycara, "Overview of CORTES: A Constraint Based Approach to Production Planning, Scheduling and Control", in *Proceedings of the Fourth International Conference in Expert Systems in Production and Operations Management*, (ed. Martin D. Goslar), Management Science Department, College of Business Administration, University of South Carolina, 1990.
7. Elleby, P., H.E. Fargher, and T.R. Addis, "Reactive Constraint-Based Job-Shop Scheduling", in *Expert Systems and Intelligent Manufacturing*, M.D. Oliff (ed.), North-Holland, New York, 1988.
8. Hynynen, J. and O. Lassila, "On the Use of Object-Oriented Paradigm in a Distributed Problem Solver", *AI Communications* 2(3) 142-151, 1989.
9. Jorysz, H. R. 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 B.V. (North-Holland), Amsterdam, 1991.
10. Kiczales, Gregor, Jim des Rivières and Daniel G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, Cambridge (Mass.), 1991.
11. Lassila, O., "Frames or Objects, or Both?" *Workshop Notes from the Eight National Conference on Artificial Intelligence (AAAI-90): Object-Oriented Programming in AI.* Boston (Massachusetts, U.S.A.), July, 1990. [Also published as report HTKK-TKO-B67, Otaniemi (Finland), Department of Computer Science, Helsinki University of Technology, 1990].
12. Lassila, O., "The Design and Implementation of a Frame System", Master's Thesis, Otaniemi (Finland), Faculty of Technical Physics, Helsinki University of Technology, 1992.
13. LePape, C. and S.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.
14. Le Pape, C., "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, January, 1993.
15. Meng, C.C. and M.Sullivan, "LOGOS: a Constraint-Directed Reasoning Shell for Operations Management", *IEEE Expert*, 6(1), 1991.
16. Miyashita, Kazuo and Katia Sycara, "Adaptive Case-Based Control of Schedule Revision", in *Intelligent Scheduling*, (eds. M. Fox and M. Zweben), Morgan Kaufmann Publishers, 1993.
17. Ow, P.S., S.F. Smith and A. Thiriez, "Reactive Plan Revision", *Proceedings AAAI-88*, St. Paul, MN, August, 1988.
18. Sadeh, Norman, "Micro-Opportunistic Scheduling: The Micro-BOSS Factory Scheduler", in *Intelligent Scheduling*, (eds. M. Fox and M. Zweben), Morgan Kaufmann Publishers, 1993.

19. Schmucker, Kurt. J, "Object-Oriented Programming for the Macintosh", Hayden Publishing Company, Hasbrock Heights, NJ, 1986.
20. Smith, S.F., 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), June, 1990.
21. Smith, S.F., N. Keng, and K. Kempf, "Exploiting Local Flexibility During Execution of Pre-Computed Schedules", in *Applications of AI in Manufacturing* (eds. D. Nau and C. Tong), MIT Press, July, 1992.
22. Smith, S.F., "Knowledge-Based Production Management: Approaches, Results, and Prospects", *Production Planning and Control*, 3(4), pp. 350-380, 1992.
23. Smith, S.F.,"OPIS: A Methodology and Architecture for Reactive Scheduling", in *Intelligent Scheduling*, (eds. M. Fox and M. Zweben), Morgan Kaufmann Publishers, 1993.
24. Smith, S.F. and K.P. Sycara, "A Constraint-Based Framework for Multi-Level Transportation Scheduling", CMU Robotics Institute Technical Report, June 1993.
25. Wirfs-Brock, Rebecca, Brian Wilkerson and Lauren Wiener, "Designing Object-Oriented Software", Prentice Hall, Englewood Cliffs, NJ, 1990.
26. Zweben, Monte, Eugene Davis and Michael Deale, "Iterative Repair for Scheduling and Rescheduling", Technical Report, NASA Ames Research Center, MS 244-17, Moffett Field, CA 94035.

ACKNOWLEDGEMENTS

The authors would like to thank the following people of fruitful discussions and advice during the design of the system described and during the preparation of this paper: Robert Aarts, Marcel Becker, Marcia Laurenson, Seppo Törmä.