Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems[1]

Claude Le Pape
ILOG S.A.
2 Avenue Galliéni – BP 85
F-94253 Gentilly Cedex

Abstract: It has been argued that the use of constraint-based techniques and tools enables the implementation of precise, flexible, efficient and extensible scheduling systems: precise and flexible as the system can take into account any constraint expressible in the constraint language; efficient inasmuch as highly optimized constraint propagation procedures are now available; extensible as the consideration of a new type of constraint may require (especially in an object-oriented framework) only an extension to the constraint system or, in the worst case, the implementation of additional decision-making modules (without needs for modification of the existing code). The following paper presents ILOG SCHEDULE, a C++ library enabling the representation of a wide collection of scheduling constraints in terms of "resources" and "activities." ILOG SCHEDULE is based on SOLVER, the generic software tool for object-oriented constraint programming developed and marketed by ILOG. SOLVER variables and constraints can be accessed from SCHEDULE activities and resources. As a result, the user of SCHEDULE can make use of SOLVER to represent specific constraints, and implement and combine the specific problem-solving strategies that are the most appropriate for the scheduling application under consideration. It is hoped — and expected — that object-oriented constraint programming tools like SCHEDULE will enable the industry to make decisive steps toward the implementation of "state of the art," highly flexible, constraint-based scheduling applications.

---

[1] In: *Intelligent Systems Engineering,* 1994, **3**, (2), pp. 55-66.

1. Scheduling and Resource Allocation Problems

Scheduling is the process of assigning **activities** to **resources** in time. It is a decision-making process — the process of determining a **schedule**. A variety of **constraints** affect this process: activity durations, release dates and due dates, precedence constraints, transfer and set-up times, resource availability constraints (shifts, down-time), and resource sharing. These constraints define the space of admissible solutions. In addition, relaxable preference constraints characterize the quality of scheduling decisions. These preferences are related to due dates, productivity, frequency of tool changes, inventory levels, overtime, etc. Since preference constraints may conflict with one another, the resolution of a scheduling problem also consists in deciding which preferences should be satisfied and to what extent others should be relaxed. In practice, however, preference constraints are often either combined into a unique evaluation (or cost) function to maximize (or minimize), or compiled into evaluation heuristics to favour candidate solutions which satisfy (or are likely to satisfy) the preferences. In both cases, the scheduling problem is eventually defined as a set of constraints to satisfy[1]. A solution to the scheduling problem is a set of compatible scheduling decisions (such as "perform activity A_2 as soon as possible after activity A_1 on machine M") which guarantee the satisfaction of the constraints.

Scheduling problems are very different one from the other:

- First, three broad families of scheduling problems can be distinguished depending on the degrees of freedom in positioning resource supply and resource demand intervals in time. In pure **scheduling** problems (e.g., job-shop machine scheduling), the capacity of each resource is defined over a number of time intervals and the problem consists of positioning resource-demanding activities over time, without ever exceeding the available capacity. In pure **resource allocation** problems (e.g., allocation of personnel to planes or trains), the demand for each resource is known beforehand and the problem consists of allocating resources in time to guarantee that the supply always equals or exceeds the demand. In **joint** scheduling and resource allocation problems, degrees of freedom exist for deciding both which activities to perform and when, and which resources to make available for these activities.

- Different environments are subjected to different constraints which more or less contribute to the complexity of the problem. For example, a factory scheduling problem may involve only machines as resources, while another may (in addition) require the consideration of the abilities of human operators.

- The size of a scheduling problem may vary from a few dozens activities to thousands of activities. For complexity reasons, the scheduling algorithms that work well for the small problems may not be applicable to the bigger problems.

- Important numerical features of scheduling problems vary from one environment to the other. In the same environment, they also vary from a problem-solving situation to the other. For example, the variation of the duration of manufacturing operations depends on the manufactured products, and the importance of bottleneck resources varies with the global load of the shop. Again, the scheduling algorithms that work well for some problems may not be applicable to other problems.

- Depending on the environment, the suitable response time for the construction of a schedule may vary from a few microseconds to a few days. Also, it may be necessary to incrementally modify the schedule, either as a response to environmental changes, or because it is more appropriate for a human to "make the decisions."

The existence of such disparities implies that rigid scheduling procedures, designed to provide optimal or near-optimal schedules in particular circumstances, are in general not satisfactorily applicable in other circumstances, either because some constraints (e.g., abilities of human operators) cannot be taken into account, or because satisfactory schedules cannot be generated in an acceptable amount of time.

There are two conceivable responses to the important variety and variability of scheduling problems. The first is to develop highly flexible systems, enabling the scheduler to combine various types of schedule formation and revision algorithms and dynamically adapt the overall scheduling strategy to the problem-solving situation at hand. Examples of flexible constraint-based scheduling systems include ISIS[2-3], OPIS[4-5], FLYPAST[6], SONIA[1, 7-9], DAS[10-12], and more recently the REDS^2

system of Hadavi et al[13]. Interesting comments about the most well-known systems and the evolution of work in this domain appear in the reviews of Anne Collinot[7], Peter Burke[10], Pauline Berry[12], Howard Beck[14], Khalil Hindi[15], Patrick Prosser[16], and Claude Le Pape[17]. The present paper argues that the emergence of object-oriented constraint programming tools brings about important opportunities for the industrial implementation of flexible constraint-based scheduling systems. This assertion is backed up with the development of SCHEDULE, a library enabling the representation of a wide collection of scheduling constraints, and with the results of preliminary experiments (not reproduced here but in previous papers[18-20]) involving the low-level primitives of the library.

A second, complementary, response is to enable the rapid development and adaptation of efficient scheduling applications, targeted to particular problems. In this perspective, a library of scheduling constraints is undoubtedly useful. However, the user of such a library must be able to (a) extend the library to enable the representation of domain-specific constraints and (b) implement and combine the specific problem-solving strategies that are the most appropriate for the considered problems. SCHEDULE is based on SOLVER, the generic software tool for object-oriented constraint programming developed and marketed by ILOG. SOLVER variables and constraints can be accessed from SCHEDULE activities and resources. As a result, the user of SCHEDULE benefits from all the flexibility and extensibility of SOLVER to develop new types of constraints and implement new problem-solving strategies.

Section 2 presents the main concepts of constraint programming and discusses the use of these concepts for the development of flexible scheduling systems. Section 3 describes **resource time-tables** as implemented in the SCHEDULE library to enable the representation of many sorts of **resource constraints**, i.e., constraints concerning the use of resources over time. Section 4 provides an overview of SCHEDULE (version 1.0) and Section 5 presents an example. Finally, Section 6 discusses the possibility of extending the SCHEDULE library.

2. Constraint-Based Reasoning

The main interest of constraint programming lies in actively using the constraints to reduce the computational effort needed to solve a problem. Constraints are used not only to test the validity of a solution, as in conventional programming languages, but also in a constructive mode to deduce new constraints and detect inconsistencies. For example, from $(x < y)$ and $(x > 8)$, we deduce, if x and y denote integers, that the value of y is at least 10. This process is called **constraint propagation**. If later we add the constraint $(y \leq 9)$, a contradiction is immediately detected. Without propagation, the test $(y \leq 9)$ could not be performed before the instantiation of y: no contradiction would be detected at this stage of the problem-solving process.

Constraints provide a specification of admissible assignments of values to variables. Domain constraints such as $(x \in \{2\ 4\ 8\ 16\})$ and $(y \in \{2\ 4\ 8\ 16\ 32\ 64\})$ describe domains over which variables can vary. Variable relations such as $(x + 1 \geq 2 * y * y)$ define a subset of the cartesian product of these domains. Variables and constraints are often represented in an hypergraph whose vertices are the variables (with the associated domains) and whose hyperedges correspond to the variable relations. In particular, constraints involving two variables are often organized in a graph, as shown in figure 1.

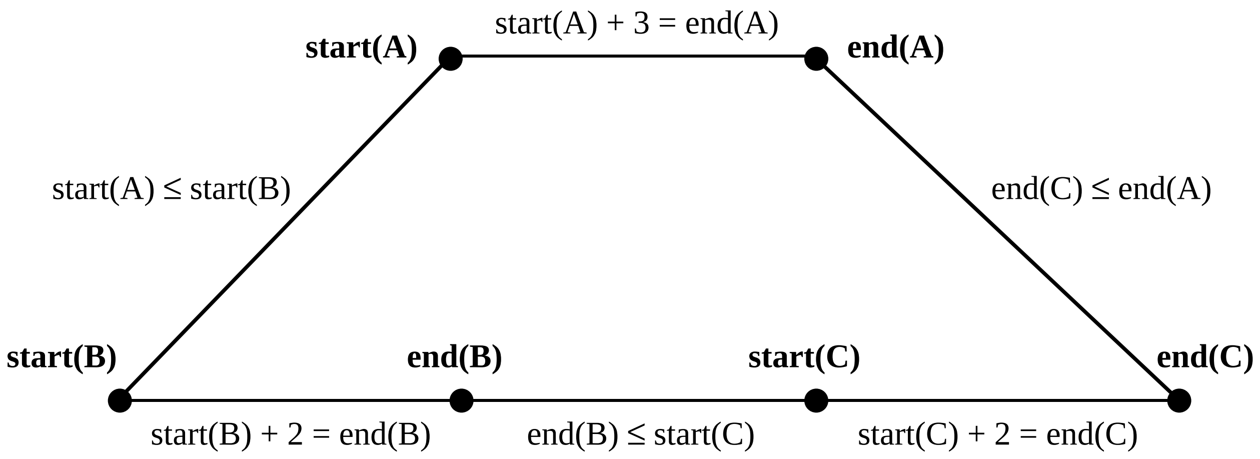

Figure 1: Graph of Variables and Constraints

Three types of programming models support constraint propagation:

- In the first type of model, generally limited to discrete problems, a constraint satisfaction problem is described as a graph. The nodes of the graph represent the problem variables, to which finite domains are associated, and the edges represent the constraints[21].

- The second type of model stems from logic programming, where constraint solving can be viewed as an extension of unification[22-30].

- Finally, a third type of model consists in distinguishing the concept of a generic constraint (a relation represented, for example, by a class in an object-oriented language) and the concept of an instance of the constraint applied to specific variables of the problem to solve[31-35]. The main advantage of this type of model is that it allows the user to define very precisely the constraints of the problem on the one hand, and to implement the most appropriate problem-solving strategy on the other hand. In addition, it permits the close integration of constraint-based methods with an object-oriented programming environment[34].

The availability of constraint propagation methods within an object-oriented environment brings numerous advantages for the development of complex scheduling applications. First, it is easy to define multiple classes of resources and attach appropriate constraint propagation methods to each class. For example, the capacity of a resource may be bounded either in terms of **power** or in terms of **energy** (or both). In the first case, the power available at any time (e.g., number of machines, number of men and women, electrical power measured in watts) is limited. In the second case, a limited amount of energy (e.g., in person-days, in watt hours) can be spent over a given time interval. Let us suppose, for example, that a resource consists of a group of *10* persons, each of which accepts to work every day (including the week-end) but globally not more than *5* days each week. Then the power available each day is *10 persons*, and the energy available each week is *50 person-days*. The use of multiple classes of resources allows the scheduling system to take both constraints into account and allow the same syntax for stating that an activity requires a resource, regardless of whether the resource capacity is defined in terms of power or in terms of energy.

Similarly, it is easy to define as many resource utilization patterns as needed for the activities to schedule. The simplest case is when an activity **requires** a fixed quantity *q* of a resource throughout its execution. But many variants are conceivable. For example, an activity may definitely **consume** a resource. And another may require *2* units of a resource (e.g., *2* human operators) when it starts, no resource during most of its execution and *2* units of the resource when it ends. The use of an object-oriented constraint-based programming tool enables the organization of a variety of resource utilization constraints as subclasses of a root "resource constraint" class. With other approaches, either code would have to be duplicated or tests would have to be made at run-time to distinguish among the different types of resource requirements.

The use of an object-oriented constraint-based programming tool also simplifies the implementation of multiple versions of a constraint. "Multiple versions" means that the semantic meaning of the constraint does not change from a version to the other, but each "version" may differ from an operational point of view. For example, one version of the constraint may support non-monotonic reasoning while another may not. The second version will propagate more efficiently and imply much less bookkeeping, but will allow only chronological backtracking in reaction to detected inconsistencies. Another version of the constraint may also propagate more than the others: the resolution of a "really hard" problem[36] may require a large amount of constraint propagation (including propagation of redundant constraints[37-38]) while the resolution of an easier problem may be more rapid if the scheduling system can make stronger decisions without propagating too much after each decision[8]. To statically impose a constraint propagation scenario, one just has to use the corresponding version of the constraint. To dynamically impose and update the scenario, one just has to create the versions of interest (instances of different constraint classes) and inhibit or activate the appropriate versions at the appropriate instants in the problem-solving process.

More generally speaking, an object-oriented constraint-based formalism endows a scheduling system with much flexibility along three dimensions: constraint representation, problem-solving control, and integration with other systems[39-40]. In the context of an object-oriented constraint programming tool, this leads to the design of different classes of constraints that correspond to different tradeoffs between precision, efficiency, flexibility and extensibility of the final scheduling application. Given a scheduling application, the selection (and possibly the implementation) of the most appropriate constraint classes is then under the responsability of the scheduling system designer, each possible selection resulting in different tradeoffs between precision, flexibility, efficiency and extensibility.

The following section focuses on the representation of resource constraints. It discusses the representations used in OPIS[4-5] and SONIA[1, 7-9], and presents the generic mechanism proposed in the SCHEDULE library.

3. Constrained Time-Tables

Two main categories of constraints occur in most scheduling problems: **temporal constraints** and **resource constraints**.

- A temporal constraint is logically expressed by a formula $(I_1 + d \leq I_2)$. I_1 and I_2 are time points, i.e., variables representing start times or end times of activities, and d a minimal delay that must elapse between the two time points. In practice, the propagation of such a constraint results in a constant update of (a) the minimal value of I_2, and (b) the maximal value of I_1, i.e., in the update of earliest and latest start and end times of activities.

- A resource constraint specifies that the execution of a given activity requires (or provides) a given amount (either a constant or a variable) of capacity from a given resource. Constraint propagation occurs in both directions: from resources to activities, in order to update the earliest and latest start and end times of activities; and from activities to resources in order to update the minimal and maximal capacities that can be used at any point in time.

The representation of scheduling and resource allocation problems often requires the representation of **time-tables** to precisely define the availability of different types of resources over time. In this respect, the modeling requirements to consider when designing a scheduling system for a given environment vary tremendously from an environment to the other.

The following cases represent three increasingly complex types of resource usage constraints that one might wish to enforce:

- Case A. The availability of each resource type varies over time (e.g., according to shifts in a factory) **independently** of the availability of the other resource types. For example, there may be six persons working in a manufacturing cell over a given weekday shift, and only three persons working over the corresponding weekend shift.

- Case B. Resource types are aggregated into more and more abstract types, typically to allow the representation of resource interchangeability. For example, there may be six persons working in a manufacturing cell during a given shift, two able to operate lathes and five able to operate milling machines. There are in this case three constraints to consider over the given shift: never more than two lathe operations in parallel; never more than five milling operations in parallel; and never more than six "lathe or milling" operations in parallel.

- Case C. The availability of a resource type at some points in time depends on the availability of other resource types at other points in time. For example, it may be the case that a polyvalent operator (able to operate both lathes and milling machines) must, over each shift, remain assigned to the same type of machine, **either** a lathe **or** a milling machine. In such a case, an additional constraint applies: if at some point during the shift the operator must be assigned to a lathe, then over the shift (s)he cannot be assigned to a milling machine, and conversely.

Consideration of these examples lead us to generalize the time-table modeling schemes previously adopted in a number of flexible scheduling systems such as OPIS[4-5] and SONIA[1, 7-9]. The solution adopted in SCHEDULE consists of a very generic framework allowing the definition of "tables of variables" as part of the SOLVER object-oriented constraint programming library (previously known as PECOS[33-34]). Semantically, a table of variables represents a variable the value of which is a function of some parameter, typically time. Hence, the availability of a resource is viewed as a variable the value of which must eventually be a function associating a quantity *v(t)* to each point in time *t*. As we shall see in Section 3.3, this allows representation of cases A, B and C above.

Section 3.1 discusses the constraint-based representations of time-tables used in OPIS and SONIA, Section 3.2 presents the generic mechanism used in SCHEDULE, and Section 3.3 provides information about its current implementations (in LE-LISP and in C++).

3.1. Constraint-Based Representations of Resource Time-Tables

A number of constraint-based representations of resource time-tables have been described in the literature. These representations achieve basically the same goal: constraint propagation through the time-table of a resource allows the scheduling system to maintain accurate time bounds (earliest and latest possible start and end times) for the activities that require the use of the resource, and consequently for the related activities.

The first significant constraint-based framework for the representation of resource time-tables was developed as part of the ISIS scheduling system[2-3]. This framework enables the specification of shifts and reservations of resources for scheduled activities at various levels of aggregation, i.e. for both individual machines and homogeneous work areas consisting of equivalent or similar machines. Stephen Smith[41] describes the first design of this framework for which a "one unscheduled order at a time" implementation was developed. "One unscheduled order at a time" means that constraint propagation is performed for a unique order to schedule, i.e., that the constraint propagation system cannot simultaneously handle two unscheduled orders. A new object-oriented design, allowing any number of unscheduled orders, was then put together by the OPIS team and implemented by Kevin Neel and Claude Le Pape as part of the OPIS system[42]. The approach used in OPIS may be summarized as follows:

- A hierarchical model is used to represent resources and activities to be performed. Schedules are developed and maintained at different levels of aggregation which are explicitly associated with resources and activities; constraint propagation is correspondingly performed at different levels.

- Scheduling constraints are attached to resources and activities, and combined to derive time bounds. A description of the original constraints that collectively impose a bound is explicitly recorded.

- Time bound constraints are maintained by an object-oriented propagation process: through messages, resources and activities communicate constraints and cooperate to compute time bounds.

- When time bounds are inconsistent, their origins provide the information required to construct an appropriate description of the conflicting situation. This description provides the scheduling system (i.e., OPIS) with information needed to make reactive decisions.

Even though this design appears particularly sound, a few remarks apply. First, the declarative semantics of the various constraints implemented in OPIS are not completely explicited. Second, resources and activities are considered in OPIS as the objects performing the propagation of the constraints. As a result, the implementation relies on particular assumptions concerning resources and activities. For example (unless recent extensions exist), the constraint management system of OPIS assumes that every activity can be interrupted if there is a break between two shifts, but cannot be pre-empted by any other activity; this assumption generally holds in industrial environments; nevertheless, the system is probably difficult to extend to cases in which the assumption does not hold. Third, as a consequence of the two other points, there is in OPIS a built-in interpretation of the aggregation levels, which makes the initial design difficult to extend to the efficient representation and propagation of complex resource constraints, such as those presented in cases B and C above.

The controllable constraint propagation system developed as part of the SONIA scheduler[1, 7-9] provides a response to the semantic remarks. In SONIA, temporal and resource reservation constraints do possess precise declarative semantics and the constraint propagation system is proven refutation-complete for disjunctive and conjunctive formulas of temporal and reservation constraints. This means it is proven that the axioms available in the system allow the detection of any inconsistency between disjunctive and conjunctive formulas of temporal and reservation constraints. Control rules are then usable to perform only part of the propagation allowed by the axioms. However, there is no explicit notion of time-table in the system. The information that corresponds to a time-table (i.e., the set of shifts and reservations made for a given resource) is split up into a number of logical formulas. As a result, the relationships between unscheduled activities and implicit time-tables must be handled explicitly, outside of the constraint propagation system (by posting *n* logical formulas instead of one constraint) and the same is true for relationships between time-tables such as those to account for in cases B and C.

Other constraint-based models of scheduling problems include explicit or implicit time-tables as well. Well-documented examples include DAS[10], COMPASS[43], REPORT[44] and TOSCA[45]. Most stochastic models of resource allocation[46-50] are also interpretable as computing "expected" time-tables. Each of these (deterministic or stochastic) models implements distinctive functionalities; but none appears generic enough to allow easy extensions toward complex resource constraints such as those of cases B and C.

3.2. Tables of Variables

The semantic model we propose for a time-table is very simple: a time-table is a constrained variable the value of which is a function associating a value *v(t)* to each point in time *t*. Two implementations are available in SCHEDULE:

- The first implementation assumes a discrete representation of time and memorizes the status of the variable (current value, current domain, current constraints) for each instant *t* in an interval *[a b)*. Information in the time-table is accessed in constant time, but the modification of the table from a date *c* to a date *d* (e.g., to reserve a resource for a given activity) requires time proportional to *(d – c)*. This type of implementation is particularly appropriate when the durations of activities are not much larger than the precision required in building the schedule.

- The second implementation does not make any assumption about the discrete or dense nature of time and memorizes the instants in time at which the status of the variable changes. Information in the time-table is accessed in time proportional to the number *n* of status changes (a slightly more complex implementation would allow an access time in *log(n)*) and a modification of the table from a date *c* to a date *d* requires time proportional to *n*. This type of implementation is particularly appropriate when there are very few activities to consider but the activities have to be positioned very precisely on the time-line.

These two implementations are referred to as **discrete array** and **sequential table**.

In addition to distinguishing discrete arrays and sequential tables, SCHEDULE allows the user to specify that the value *v(t)* is constant over intervals of a given size *g*, called the **grain** of the table. The default value of *g* is *1*. The ratio between the schedule **horizon** (the duration of the overall time period to schedule) and the **grain** is a good indicator of the interest of a discrete array compared to a sequential table: when the ratio is small (e.g., a month in days), a discrete array is more appropriate; on the opposite, when the ratio is large (e.g., a month in seconds), a sequential table constitutes the best representation.

One of the main advantages of the underlying theoretical model is that it allows the management of discrete arrays and sequential tables of any type of variable: a discrete array of integer variables is built from a prototype integer variable; a sequential table of floating point variables is built from a prototype floating point variable; etc. In an object-oriented framework, it is therefore straightforward to implement a class of array and a class of table for each existing class of variable. For example, SOLVER contains mainly five types of constrained variables:

- constrained integer variables: a table of integer variables may represent the time-varying capacity of a discrete resource (e.g., the number of milling machines available or in use at any time);
- constrained floating point variables, to which a degree of precision is associated: a table of floating point variables may represent the time-varying capacity of a continuous resource (e.g., the amount of water available or in use at any time, if an integer representation is not sufficient);
- constrained Boolean variables: a table of Boolean variables may represent the time-varying

capacity of a unary resource (e.g., a specific person);

- constrained enumerated variables, the values of which are objects of any type: a table of enumerated variables may represent the state of the resource at any time (e.g., the value may belong to *{on off maintenance}* or point to a type of activity being performed);

- constrained set variables, allowing the expression of constraints on sets: a table of set variables may represent the set of activities being performed with a resource, or the set of sub-resources available at any time.

In this respect, the model significantly extends the functionalities of time-tables as used in previous constraint-based scheduling systems.

One of the main interests of time-tables as tables of variables is that they can be used to propagate constraints in a very generic fashion. An interesting case in scheduling consists in using the information available in time-tables to refine the time-bounds of activities to be performed. A generic class of constraints is available for this purpose. For example, assume that a table of set variables represents a set of sub-resources available at any time, and that some activity is executable only when some particular sub-resource is present. To ensure that the time-bounds of the activity are always updated with respect to the contents of the table, one merely has to create a subclass of the generic constraint class, specify as a method of the subclass the test that a set variable must verify for the activity to be executable, and create an instance of the subclass. This does not make more than one page of code.

Numeric capacity constraints (cases in which the condition to satisfy for an activity to be executable is the availability of some amount of resources) are predefined for both **power** and **energy** interpretations of capacity time-tables. This means the capacity time-table may either specify that a limited amount of resources (in persons, in watts) are available at any time or indicate that a limited amount of energy (in human-months, in watt hours) can be spent over a given time interval (the size of which is the grain of the table). Also, one can use the capacity constraints not only to update time bounds of activities, but also to automatically reserve the capacity from the table when activities are constrained to perform between given dates.

Finally, many types of constraints defined on a given class of variables (e.g., ordering and arithmetic constraints for integer variables) are directly transposable to the corresponding classes of arrays and tables. Semantically, posing a constraint on an array or a table of variables is equivalent to posing the constraint for every *t* in the index range *[a b)* of the table. For example, if A_1 and A_2 are two arrays of integer variables, the constraint $(A_1 < A_2)$ means that for every *t*, the value $v_1(t)$ corresponding to array A_1 must be strictly smaller than the value $v_2(t)$ corresponding to array A_2. As we shall see in the next section, such functionalities are particularly useful for scheduling problems involving complex constraints, such as those of cases B and C.

3.3. Implementation

Tables of variables have been implemented on top of the two versions of SOLVER, the generic software tool for object-oriented constraint programming marketed by ILOG. SOLVER is a library of object classes, functions and control structures, available in the LE-LISP or (depending on the needs of the application) C++ programming language. SOLVER extends the object-oriented programming concept of the two languages by providing an additional library for constraint definition and management. Thus, SOLVER represents the constraint management component of an application, used to implement the combinatorial part of the application, and perfectly integrated with the other computing processes (graphical interface, connection to databases, etc.) because they rely on the same data structures (objects). In SOLVER, constraint propagation enforces arc-consistency, using an algorithm similar to AC-5[51].

A preliminary implementation of tables of variables (cf. Section 3.2) was first made in LE-LISP to test the design ideas and evaluate the cost of genericity on a problem which did not require the full functionalities of generalized time-tables. This effort followed previous work on the considered problem[18] and showed the overhead cost to be reasonably low (33% on average on CPU time, decreasing with the size of the problem)[19-20]. A second, more complete, version was then developed in C++. This version includes more functionalities than the LE-LISP version, in particular to deal with complex constraints such as those encountered in cases B and C.

Case B is handled by adding together (a) the table representing the number of lathe operators working at any time *t* and (b) the table representing the number of milling machine operators working at any time *t*. The sum is a table which represents the number of operators working in the manufacturing cell at any time *t*; its variables are constrained to be no greater than six over the considered shift.

```
cellCapacityTable = CtAdd(latheCapacityTable,
                          millingCapacityTable);
CtIntVarArrayIterator iterator(cellCapacityTable,
                               shiftStartTime,
                               shiftEndTime);
while (iterator.next(variable))
    variable->setMax(6);
```

Case C is represented by defining either additional discrete arrays with grain greater than *1* (if all shifts have the same duration) or additional non-decomposable sequential tables (decomposed in given predetermined shifts). These new tables (*aggregateLatheCapacityTable* for lathe operators and *aggregateMillingCapacityTable* for milling machine operators) represent the number of operators assigned to lathes (respectively milling machines) over each shift. These numbers are greater than the numbers of operators actually processing goods on lathes (respectively milling machines) at any point in time during the shifts. The following code will work, even if the lathe shifts do not coincide with the milling shifts.

```
CtGe(aggregateLatheCapacityTable,
     latheCapacityTable);
CtGe(aggregateMillingCapacityTable,
     millingCapacityTable);
cellCapacityTable = CtAdd(aggregateLatheCapacityTable,
                          aggregateMillingCapacityTable);
CtIntVarArrayIterator iterator(cellCapacityTable,
                               shiftStartTime,
                               shiftEndTime);
while (iterator.next(variable))
    variable->setMax(6);
```

4. SCHEDULE 1.0: An Overview

The SCHEDULE library proposes a simple object model for the **representation** of scheduling and resource allocation problems in terms of "resources" and "activities." The model consists of a series of C++ classes and functions that implement the concepts of "resource" and "activity" in terms of SOLVER variables and constraints.

For the **resolution** of the represented problem, the user of SCHEDULE relies on SOLVER, in order to implement problem-solving strategies that are appropriate for the scheduling problems under consideration. Also, the user relies on SOLVER when it is necessary to extend the problem representation to account for domain-specific constraints such as "the duration of the cooling activity equals twice the duration of the preceding heating activity" or "the preparation of the caper sauce starts when the vegetables are half-cooked." For this purpose, SOLVER variables and constraints can be accessed from SCHEDULE activities and resources. The expression `act->getStartVariable()`, for example, returns a SOLVER variable representing the start time of the activity `act`.

4.1. Schedules

The object model of SCHEDULE consists of a number of object classes allowing the representation of a scheduling problem. The first object to create is an instance of the `CtSchedule` class. A `CtSchedule` is a global object which represents the schedule. Most applications include only one instance of the `CtSchedule` class. However, multiple instances of the `CtSchedule` class may be used, for example to simulate decentralized distributed scheduling. A `CtSchedule` object includes, among its data members, a set of activities and a set of resources.

The constructor of a `CtSchedule` instance accepts two arguments `timeMin` and `timeMax`, which define the time interval covered by the schedule. In the following, `timeMin` and `timeMax` are referred to as the **time origin** and as the **time horizon** of the schedule. The time origin and the time horizon are used "by default" to initialize time-tables of resources as well as earliest start times and latest end times of activities.

4.2. Resources

Four classes of resources, `CtDiscreteResource`, `CtUnaryResource`, `CtDiscreteEnergy`, and `CtStateResource`, are defined in SCHEDULE 1.0. All of these classes inherit from the `CtResource` class, which is an abstract class, i.e., a class with no direct instance.

4.2.1. Discrete Resources

A `CtDiscreteResource` represents a resource of discrete capacity. Capacity varies with time: at any time *t*, it represents the number of copies or instances of the resource that are available (e.g., the number of milling machines available in a manufacturing shop, the number of bricklayers at work on a construction site). By discrete, it is meant that capacity is defined to be a positive integer. Each activity may require or provide some amount (e.g., one milling machine, three bricklayers) of the resource capacity. An argument to the constructors of `CtDiscreteResource` instances enables the distinction between the resources "to be required" and the resources "to be provided" by activities.

Three notions of capacity are defined. The **theoretical capacity** of a discrete resource is a bound on the amount of capacity that can be used or provided at any point in time. This notion can be opposed to the **maximal capacity** that can be used or provided "in practice" at a particular point in time or over a particular interval of time: the maximal capacity typically varies over time, while the theoretical capacity is an intrinsic property of the resource. Let us note that the theoretical capacity can be infinite. Also, at any point in time, the maximal capacity cannot exceed the theoretical capacity.

It is also possible to constrain the capacity used or provided to exceed some **minimal capacity** over some interval of time. This is typically useful in the case of "provided" resources: the constraint enables to specify that some minimal amount of resource capacity must be provided. An inconsistency is detected if at any point in time the minimal capacity exceeds the maximal capacity.

When finite, the theoretical capacity is passed as an argument to the constructor of the `CtDiscreteResource` instance. This means the theoretical capacity of a resource must be known when the corresponding `CtDiscreteResource` object is created. On the other hand, the maximal (respectively minimal) capacity can be updated at any time by calling the `setCapacityMax` (respectively `setCapacityMin`) function: if `res` is a pointer to a `CtDiscreteResource`, the expression `res->setCapacityMax(t1, t2, c)` signifies that from time `t1` to time `t2`, the capacity that is used or provided cannot exceed `c`.

It is also possible to specify that the capacity that is used or provided is allowed to change only at times of the form `(timeMin + i * timeStep)`. This is useful for example if the resource is a group of consulting engineers: if an engineer expects to visit a customer on a particular day, (s)he would rather be considered "required" for the whole day, even if the visit is expected to last a few hours.

4.2.2. Unary Resources

A `CtUnaryResource` represents a resource whose capacity is one. There are two methods to take into account the constraints concerning the requirement or the provision of a unary resource. The first method is a specialization of the method used for discrete resources. It allows capacity to vary with time: at any time *t*, the resource may or may not be available. In contrast, the second method deals only with requiring (or providing) activities: it consists of updating the earliest and latest start and end times of activities to ensure that the time intervals over which two activities require (or provide) the same unary resource cannot overlap.

These two methods correspond in fact to two different formulations of the capacity constraints:

- The **cumulative formulation** states that the capacity required by all activities at any point in time *t* cannot exceed the capacity available at time *t*.
- The **disjunctive formulation** states that for any two activities A_i and A_j which require the same unary resource, either A_i must precede A_j, or A_j must precede A_i.

The disjunctive formulation automatically permits the deduction of many relevant pieces of information (via constraint propagation). Indeed, in this case, it is sufficient to prove that A_i cannot precede A_j to deduce that A_j must precede A_i, and vice versa. The propagation of the disjunctive constraint tends to result in more information being deduced, which results in greater efficiency when searching for solutions.

On the other hand, the disjunctive formulation assumes that the capacity of the resource remains stable over time (i.e., that the resource is always available), which leads to the introduction of "fake" activities to represent existing variations of capacity over time. The introduction of these "fake" activities inevitably results in a reduction of the efficiency of the constraint propagation process.

In practice, each of the two formulations proves more efficient than the other on some problems, and less efficient than the other on other problems. For each unary resource, the user of SCHEDULE specifies the formulation to be used.

4.2.3. Discrete Energies

A `CtDiscreteEnergy` is similar to a `CtDiscreteResource` but its **energetic** capacity — as opposed to **instantaneous** capacity — is defined with respect to given time intervals (e.g., days, months, years) as the amount (e.g., in watt hours, in human-months) that can be made available over those intervals.

4.2.4. State Resources

A `CtStateResource` represents a resource the state of which can vary over time. Each activity may, throughout its execution, require a state resource to be in a given state (or in any of a given set of states). Consequently, two activities may not overlap if throughout their execution they require incompatible states.

Specific constraints can easily be added to specify that modifying the state of the resource requires some amount of time, e.g., that d_{ij} units of time are necessary to change the state of the resource from state S_i to state S_j. This enables the representation of set-up constraints.

4.3. Activities

A unique class of activities, `CtIntervalActivity`, is defined in SCHEDULE 1.0. This class inherits from the `CtActivity` class.

4.3.1. Interval Activities

A `CtIntervalActivity` is an activity which executes without interruption from its start time to its end time, and requires (or provides) the same resources from the beginning to the end of its execution. Other classes of activities may be defined by combining instances of the `CtIntervalActivity` class.

An activity is defined by its start time, end time and duration. Each of these three parameters may be fixed or constrained to lie between a minimal and a maximal value. In particular, the duration of the activity may be either a constant (passed to the constructor of the `CtIntervalActivity` instance) or a variable. If `act` is a pointer to a `CtIntervalActivity`, the expressions `act->getStartMin()`, `act->getStartMax()`, `act->getEndMin()`, `act->getEndMax()`, `act->getDurationMin()` and `act->getDurationMax()` return (respectively) the earliest start time, the latest start time, the earliest end time, the latest end time, the smallest possible duration and the longest possible duration of the `CtIntervalActivity`. A number of functions are defined to update these parameters. In particular, the three expressions `act->setStartTime(startTime)`, `act->setEndTime(endTime)` and `act->setDuration(duration)` enable the instantiation of the start time, end time and duration of an activity to the given values (respectively `startTime`, `endTime` and `duration`).

Activities can be linked together by a number of temporal constraints. For example, the expression `act2->startsAfterEnd(act1, delay)` signifies that `act2` cannot start before the end of `act1`. (In other terms, `act1` precedes `act2`.) In addition, at least the given `delay` must elapse between the end of `act1` and the beginning of `act2`. The `startsAfterEnd` function can be invoked with a negative `delay`. In this case, it means that `act2` can start before the end of `act1`, but the difference between the end time of `act1` and the start time of `act2` cannot exceed `–delay`. In other terms, the expression `act2->startsAfterEnd(act1, delay)` imposes `(start(act2) ≥ end(act1) + delay)`.

Activities **require**, **provide**, **consume** and **produce** resources:

- An activity **requires** a resource if some amount of the resource capacity must be made available for the execution of the activity. The capacity is **recoverable**: when the requiring activity ends, the required capacity can be allocated to other activities.

- An activity **provides** a resource if some amount of the resource capacity is made available through the execution of the activity. The capacity is **recoverable**: when the providing activity ends, the provided capacity is no longer available.

- An activity **consumes** a resource if some amount of the resource capacity must be made available for the execution of the activity and the capacity is **non-recoverable**. When the consuming activity ends, the consumed capacity is no longer available.

- An activity **produces** a resource if some amount of the resource capacity is made available through the execution of the activity and the capacity is **non-recoverable**. When the producing activity ends, the produced capacity is definitely available — until another activity consumes it!

For example, `act->requires(res, c)` signifies that the activivity `act` requires the amount of capacity `c` of the resource `res`. Let us note that `c` may be either a constant or a variable. This allows (for example) the representation of the situation in which the duration of an activity depends on the amount of resources assigned to the activity: the capacity amount `c` and the duration `d` are in this case two variables, linked together via a domain-specific constraint `(d = f(c))`.

Each resource (instance of the `CtResource` class) maintains the sets of its requiring activities, providing activities, consuming activities and producing activities. Constraints are posted to guarantee the satisfaction of the resource capacity limitations. These constraints propagate in both directions: from the resource to the activities, in order to update activity time-bounds (earliest and latest start and end times); and from the activities to the resource, in order to update the minimal and maximal capacities that can be used at any point in time.

5. An Example

The aim of this section is to illustrate the use of the SCHEDULE library on a simple example. The example consists of a number of tasks to perform prior to move in a new house (cf. figure 2). Each task (`masonry`, `carpentry`, ...) is assigned a given duration (aside to the task in the graph) and assumed to cost a thousand dollars a day, to be paid at the beginning of the task. This means seven thousand dollars must be paid for `masonry` to begin, three thousand dollars must be paid for `carpentry` to begin, etc. Arcs in the graph represent precedence constraints. For example, `masonry` must be finished for `carpentry` to begin. The goal is to move in as soon as possible, given that twenty thousand dollars are currently available, and nine thousand dollars will become available in fifteen days.

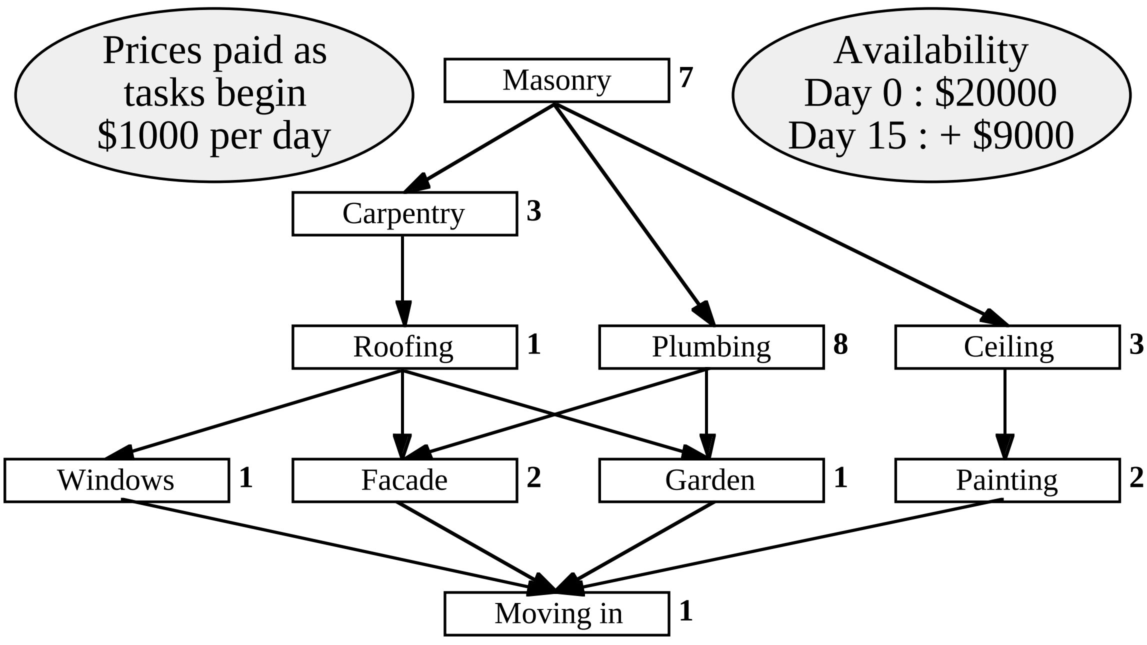

Figure 2: Scheduling Problem

Scheduling with precedence constraints and consumable resources (exclusively) is known to be a polynomial problem. This means the resolution of this problem does not necessitate exploration of the search space. The problem was voluntarily chosen here to illustrate the use of SCHEDULE without requiring acquaintance with the SOLVER library. As mentioned in Section 4, the user of SCHEDULE would, in general, use SCHEDULE for the representation of the scheduling problem, and SOLVER for the constraint-based exploration of the search space.

What is interesting in this example is that the use of SCHEDULE guarantees the update, after each scheduling decision, of the earliest and latest start times of all the tasks, with respect to both the precedence and the consumed resource constraints. As a result, the following algorithm is guaranteed to provide the optimal solution: while some task is unscheduled (i.e., does not have a fixed start time), select the most urgent unscheduled task (i.e., the task with the smallest latest start time) and schedule it as early as possible (i.e., fix its start time to its earliest start time). Using SCHEDULE, this translates into the following code:

```
CtBoolean IsUnScheduled(CtActivity* act) {
  // Returns true if act does not have a fixed start time.
  if (act->getStartVariable()->isBound())
    return CtFalse;
  else
    return CtTrue;
}

CtBoolean IsMoreUrgent(CtActivity* act1,
                       CtActivity* act2) {
  // Returns true if act1 is more urgent than act2.
  // Returns true if act2 is unbound (== 0).
  if (act2 == 0)
    return CtTrue.
  else if (act1->getStartMax() < act2->getStartMax())
    return CtTrue;
  else
    return CtFalse;
}
```

```
CtActivity* SelectActivity(CtSchedule* schedule) {
  // Returns the unscheduled activity with the smallest latest
  // start time. Returns 0 if all activities are scheduled.
  CtActivity* bestActivity = 0;
  // Creates an iterator to iterate on all activities.
  CtActivityIterator* iterator(schedule);
  CtActivity* newActivity;
  while (iterator.next(newActivity))
    if ((IsUnScheduled(newActivity))
        && (IsMoreUrgent(newActivity, bestActivity)))
      bestActivity = newActivity;
  return bestActivity;
}

void SolveProblem(CtSchedule* schedule) {
  // Solves the problem assuming constraints have been posted.
  CtActivity* act = SelectActivity(schedule);
  while (act != 0) {
    act->setStartTime(act->getStartMin());
    act = SelectActivity(schedule);
  }
}
```

The algorithm above assumes that all constraints have been posted. This includes creating a `CtSchedule` with appropriate time origin and time horizon, creating each activity with a fixed duration, posting the precedence constraints, creating the budget resource, and posting the resource constraints. The following types of statements are used for these purposes:

```
// To create a schedule with origin 0 and given horizon.
CtSchedule* schedule =
  new CtSchedule(0, horizon);

// To create an activity with the given duration.
CtIntervalActivity* act =
  new CtIntervalActivity(schedule, duration);

// To post a precedence constraint between act1 and act2.
act2->startsAfterEnd(act1, 0);
```

```
// To create a total budget of limited capacity (here 29000).
CtDiscreteResource* res =
  new CtDiscreteResource(schedule,
                         CtRequiredResource,
                         capacity);

// To state that only cap (here 20000) is available prior to a
// given date (here 15).
res->setCapacityMax(0, date, cap);

// To state that an activity act consumes c units ($) of res.
act->consumes(res, c);
```

Using these six types of statements, one can easily represent the constraints of any scheduling problem with precedence constraints and consumable resources. Then the function `SolveProblem` can be invoked to determine the optimal solution to the problem. Figure 3 presents the optimal solution to the problem represented in figure 2.

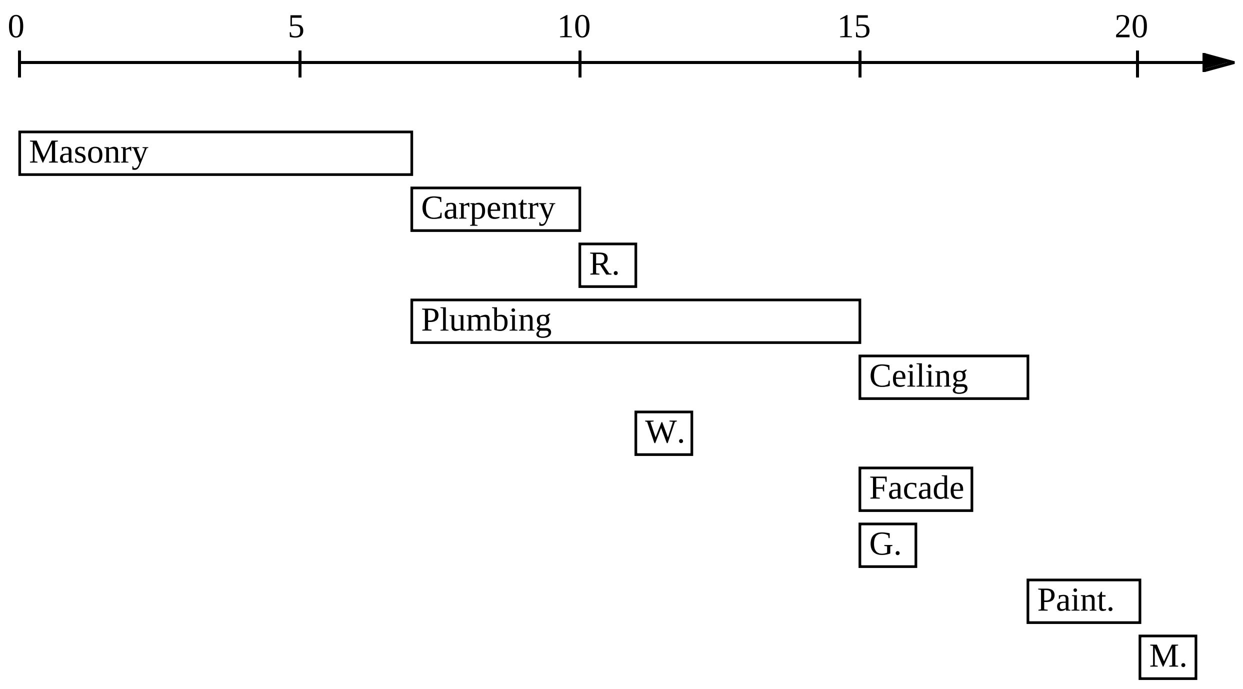

Figure 3: Optimal Solution to the Scheduling Problem

6. Possible Extensions of the Library

There are three important facts concerning the extensibility of the SCHEDULE library: (1) SCHEDULE is built on top of SOLVER; (2) the user of SCHEDULE can access the underlying SOLVER model (variables and time-tables of variables); and (3) SOLVER **is** extensible, i.e. enables the user to define new types of constraints and new types of problem-solving strategies with respect to the requirements of a specific application. As a result, any user of SCHEDULE can extend the library to represent additional types of constraints and implement specific problem-solving strategies.

Similarly, the fact that SCHEDULE and SOLVER are C++ librairies (and not new programming languages) facilitates the integration of constraint-based scheduling algorithms with other parts of a scheduling application such as a Gantt chart graphical editor, a database management system, or a rule-based execution monitoring system. The use of other C++ tools provided by ILOG (such as VIEWS for the implementation of graphical interfaces, DBLINK for communication with relational databases, RULES[52] for the development of an execution monitoring system, or BROKER[53] for the development of distributed scheduling applications) may even facilitate the development of these "other parts". But it is not compulsory: any other software tool can be used provided that it communicates easily with C++ or C. In this respect again, the user of SCHEDULE can *de facto* perform the extensions needed for his or her application.

7. Conclusion

The aim of this paper was to present SCHEDULE, a constraint-based library for the representation of scheduling and resource allocation problems. SCHEDULE relies on a generic mechanism allowing the representation of tables of variables over time. This mechanism, which significantly generalizes those of OPIS[4-5, 42] and SONIA[1, 7-9], is implemented on top of SOLVER, the object-oriented constraint programming tool developed and marketed by ILOG.

Preliminary experiments with tables of variables have shown the CPU overhead due to genericity to be very reasonable (33% on average over the experiments, decreasing with the size of the problem) compared to the benefits in terms of simplicity and compactness of the application code[19-20]. In most cases, the use of SCHEDULE allows the size of the source code implementing a scheduling algorithm to be divided by 2, starting from an already compact SOLVER implementation. This constitutes a significant advantage from both a software engineering and a problem-solving viewpoint. It is hoped — and expected — that object-oriented constraint programming tools like SCHEDULE will enable the industry to make decisive steps toward the implementation of "state of the art," highly flexible, constraint-based scheduling systems.

8. Acknowledgements

I would like to acknowledge the contribution of Jean-François Puget, Younès Alaoui, Bruno Lévy and Patrick Albert to the design and development of the SCHEDULE library. Also I would like to thank the referees for their numerous remarks and suggestions on a previous version of this paper.

9. References

[1] LE PAPE, C.: 'Scheduling as Intelligent Control of Decision-Making and Constraint Propagation'. To appear in ZWEBEN, M., and FOX, M. (editors): 'Intelligent Scheduling' (Morgan Kaufmann, 1994)

[2] FOX, M. S.: 'Constraint-Directed Search: A Case Study of Job-Shop Scheduling'. PhD Thesis, Carnegie-Mellon University, 1983.

[3] FOX, M. S., and SMITH, S. F.: 'ISIS: A Knowledge-Based System for Factory Scheduling', *Expert Systems*, 1984, **1**, (1), pp. 25-49.

[4] SMITH, S. F., OW P. S., LE PAPE, C., MCLAREN, B., and MUSCETTOLA, N.: 'Integrating Multiple Scheduling Perspectives to Generate Detailed Production Plans'. Proceedings of the SME Conference on Artificial Intelligence in Manufacturing, Long Beach, California, 1986.

[5] SMITH, S. F.: 'A Constraint-Based Framework for Reactive Management of Factory Schedules'. Proceedings of the First International Conference on Expert Systems and the Leading Edge in Production Planning and Control, Charleston, South Carolina, 1987.

[6] MOTT, D. H., CUNNINGHAM, J., KELLEHER, G., and GADSDEN J. A.: 'Constraint-Based Reasoning for Generating Naval Flying Programmes', *Expert Systems*, 1988, **5**, (3), pp. 226-246.

[7] COLLINOT, A.: 'Le problème du contrôle dans un système flexible d'ordonnancement'. PhD Thesis, University Paris VI, 1988 (in French).

[8] COLLINOT, A., and LE PAPE, C.: 'Adapting the Behavior of a Job-Shop Scheduling System', *International Journal for Decision Support Systems,* 1991, **7**, (3), pp. 341-353.

[9] LE PAPE, C.: 'Des systèmes d'ordonnancement flexibles et opportunistes'. PhD Thesis, University Paris XI, 1988 (in French).

[10] BURKE, P.: 'Scheduling in Dynamic Environments'. PhD Thesis, University of Strathclyde, 1989.

[11] PROSSER, P.: 'Distributed Asynchronous Scheduling'. PhD Thesis, University of Strathclyde, 1990.

[12] BERRY, P. M.: 'A Predictive Model for Satisfying Conflicting Objectives in Scheduling Problems'. PhD Thesis, University of Strathclyde, 1991.

[13] HADAVI, K., HSU, W.-L., CHEN, T., and LEE, C.-N.: 'An Architecture for Real-Time Distributed Scheduling', *AI Magazine*, 1992, **13**, (3), pp. 46-56.

[14] BECK, H.: 'An Overview of AI Scheduling in the UK'. Technical Report, University of Edinburgh, 1992.

[15] HINDI, K. S.: 'An Overview of Knowledge-Based Industrial Scheduling'. Proceedings of the IFAC Workshop on Computer-Integrated Manufacturing in Process and Manufacturing Industries, Espoo, Finland, 1992.

[16] PROSSER, P.: 'The future of scheduling – DAI?'. Proceedings of the IEE Colloquium on Advanced Software Technologies for Scheduling, London, United Kingdom, 1993.

[17] LE PAPE, C.: 'Programmation par contraintes et ordonnancement : historique et perspectives'. Tutorial of the Ninth Congress on Pattern Recognition and Artificial Intelligence, Paris, France, 1994 (in French).

[18] LE PAPE, C.: 'Using Object-Oriented Constraint Programming Tools to Implement Flexible "Easy-to-use" Scheduling Systems'. Proceedings of the NSF Workshop on Intelligent, Dynamic Scheduling for Manufacturing, Cocoa Beach, Florida, 1993.

[19] LE PAPE, C.: 'A Universal Constraint-Based Representation of Time-Tables: Benefits and Costs ... and Benefits'. Proceedings of the AAAI-SIGMAN Workshop on Knowledge-Based Production Planning, Scheduling and Control, IJCAI, Chambéry, France, 1993.

[20] LE PAPE, C.: 'The Cost of Genericity: Experiments with Constraint-Based Representations of Time-Tables'. Proceedings of the Sixth International Conference on Software Engineering and its Applications, Paris-La Défense, France, 1993.

[21] KUMAR, V.: 'Algorithms for Constraint Satisfaction: A Survey', *AI Magazine*, 1992, **13**, (1), pp. 32-44.

[22] AIT-KACI, H., and PODELSKI, A.: 'Is there a Meaning to LIFE?'. PRL Technical Report, Digital Equipment Corporation, 1991.

[23] CODOGNET, P., FAGES, F., and SOLA, T.: 'A Cheap Implementation of CLP(F) and its Combination with Intelligent Backtracking'. Technical Report, Laboratoire Central de Recherches Thomson-CSF, 1991.

[24] COLMERAUER, A.: 'Opening the Prolog III Universe', *Byte*, 1987, **12**, (9), pp. 177-182.

[25] COLMERAUER, A.: 'An Introduction to Prolog III', *Communications of the ACM*, 1990, **33**, (7), pp. 69-90.

[26] HAVENS, W. S., SIDEBOTTOM, S., SIDEBOTTOM, G., JONES, J., CUPERMAN, M., and DAVISON, R.: 'Echidna Constraint Reasoning System: Next-generation Expert System Technology'. Technical Report, Simon Fraser University, 1990.

[27] HAVENS, W. S.: 'Intelligent Backtracking in the Echidna CLP Reasoning System'. Technical Report, Simon Fraser University, 1991.

[28] JAFFAR, J., and LASSEZ, J.-L.: 'Constraint Logic Programming'. Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, 1987.

[29] SIDEBOTTOM, G., and HAVENS, W. S.: 'Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming'. Technical Report, Simon Fraser University, 1991.

[30] VAN HENTENRYCK, P.: 'Constraint Satisfaction in Logic Programming' (MIT Press, 1989)

[31] CASEAU, Y.: 'Abstract Interpretation of Constraints on Order-Sorted Domains'. Bellcore Technical Memorandum, 1991.

[32] LIU, B., and KU, Y.-W.: 'ConstraintLisp: An Object-Oriented Constraint Programming Language', *ACM SIGPLAN Notices*, 1992, **27**, (11), pp. 17-26.

[33] PUGET, J.-F., and ALBERT, P.: 'PECOS: programmation par contraintes orientée objets', *Génie logiciel et systèmes experts*, 1991, **23**, pp. 100-105 (in French).

[34] PUGET, J.-F.: 'Programmation par contraintes orientée objet'. Proceedings of the Twelfth International Workshop on Expert Systems and Applications, Avignon, France, 1992 (in French).

[35] SISKIND, J., and MCALLESTER D.: 'Non deterministic Lisp as a substrate for Constraint Logic Programming'. Proceedings of the Eleventh National Conference on Artificial Intelligence, Washington, District of Columbia, 1993.

[36] CHEESEMAN, P., KANEFSKY, B., and TAYLOR, W. M.: 'Where the Really Hard Problems Are'. Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, Sydney, Australia, 1991.

[37] CASEAU, Y., GUILLO, P.-Y., and LEVENEZ, E.: 'A Deductive and Object-Oriented Approach to a Complex Scheduling Problem'. Proceedings of the International Conference on Deductive Object-Oriented Databases, Phoenix, Arizona, 1993.

[38] LE PAPE, C., PUGET, J.-F., COLONEL MOREAU, and DARNEAU P.: 'PMFP: The Use of Constraint-Based Programming for Predictive Personnel Management'. Proceedings of the Eleventh European Conference on Artificial Intelligence, Amsterdam, The Netherlands, 1994.

[39] LE PAPE, C.: 'Classification of Scheduling Problems and Selection of Corresponding Constraint-Based Techniques'. Proceedings of the IEE Colloquium on Advanced Software Technologies for Scheduling, London, United Kingdom, 1993.

[40] LE PAPE, C.: 'Implementation and Integration of Different Classes of Resource Constraints for Activity Scheduling: Principles and Tradeoffs'. Proceedings of the Twelfth UK Planning Special Interest Group Meeting, Cambridge, United Kingdom, 1993.

[41] SMITH S. F.: 'Exploiting Temporal Knowledge to Organize Constraints'. Technical Report, Carnegie-Mellon University, 1983.

[42] LE PAPE, C., and SMITH, S. F.: 'Management of Temporal Constraints for Factory Scheduling'. Technical Report, Carnegie-Mellon University, 1987.

[43] FOX., B. R.: 'Chronological and Non-Chronological Scheduling'. Proceedings of the First Annual Conference on Artificial Intelligence, Simulation and Planning in High Autonomy Systems, Tucson, Arizona, 1990.

[44] LOPEZ, P.: 'Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources'. PhD Thesis, University Paul Sabatier, 1991 (in French).

[45] BECK, H.: 'Constraint Monitoring in TOSCA'. Working Papers of the AAAI Spring Symposium on Practical Approaches to Planning and Scheduling, Stanford, California, 1992.

[46] MUSCETTOLA, N., and SMITH, S. F.: 'A Probabilistic Framework for Resource-Constrained Multi-Agent Planning'. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy, 1987.

[47] SADEH, N.: 'Look-Ahead Techniques for Micro-Opportunistic Job-Shop Scheduling'. PhD Thesis, Carnegie-Mellon University, 1991.

[48] SYCARA, K. P., ROTH, S. F., SADEH, N., and FOX, M. S.: 'Resource Allocation in Distributed Factory Scheduling', *IEEE Expert*, 1991, **6**, (1), pp. 29-40.

[49] BERRY, P. M.: 'SCHEDULING: A Problem of Decision-Making Under Uncertainty'. Proceedings of the Tenth European Conference on Artificial Intelligence, Vienna, Austria, 1992.

[50] BERRY, P. M.: 'The PCP: A Predictive Model for Satisfying Conflicting Objectives in Scheduling Problems', *Artificial Intelligence in Engineering*, 1992, **7**, (4), pp. 227-242.

[51] VAN HENTENRYCK, P., DEVILLE, Y., and TENG, C. M.: 'A General Arc-Consistency Algorithm and its Specializations', *Artificial Intelligence*, 1992, **57**, (3), pp. 291-321.

[52] ALBERT, P., and FAGES, F.: 'XRETE : un outil pour les systèmes experts temps réel', *Génie logiciel et systèmes experts*, 1992, **28**, pp. 22-34 (in French).

[53] CEUGNIET, X., FORNARINO, C., and LEXTRAIT, V.: 'Gestion de la cohérence dans les systèmes orientés objets distribués'. Proceedings of the Sixth International Conference on Software Engineering and its Applications, Paris-La Défense, France, 1993 (in French).