Appears in *Proc. of PAAM'98, March 1998, London, U.K.*

Agent-Based Decision Support for Managing Print Tasks

J. M. Andreoli, U. M. Borghoff, S. Castellani, R. Pareschi
Xerox Research Centre Europe, Grenoble, France
G. Teege
Institut für Informatik, Technische Universität München, Germany

Abstract

We consider here a typical instance of a non-trivial print tasks management problem, and we show how to handle it within Zippin, a highly interactive constraint-based environment for scheduling and decision support. The aim of Zippin is to provide the user with synthetic views of an ongoing scheduling process from multiple perspectives, allowing the user to navigate through the space of possible schedules and to make informed decisions. The navigation space is defined by a system of constraints accessed by the user through the views and by a set of software agents, which manipulate the constraints store. Constraints identify goals that agents have to fulfill as well as effects that their actions propagate into the environment. Thus, the agent model implements a type of behavior which is *teleological* (agents have intentions that guide their actions) and *stigmergic* (agents communicate through the effects that their actions produce on the environment). We also describe how our model may be extended with agents which implement heuristic actions handling specific scheduling problems.

Key Words. multi-agent systems, implicit coordination, print task scheduling, simulation.

1 Introduction

Intelligent scheduling and coordination of print tasks has become a major challenge for print-shops using more and more sophisticated and dedicated pieces of hardware, possibly distributed over several autonomous sites. As with many production management scheduling problems [15], entirely automated solutions have proven difficult to implement in a real life environment, because of the huge number of parameters an automatic system would have to deal with, and also because these parameters and the constraints which relate them are not always easy to explicit. Instead, interaction with field workers is of crucial importance to guide a computer assisted scheduling process.

In fact, the kind of scheduling we consider here is not limited to assigning begin and end dates to a given set of tasks in order to satisfy some constraints or to optimize some cost function. It is also concerned with determining *which* sets of tasks are capable of achieving a given goal, respecting the constraints and to support decision-makers to choose among alternative solutions. In the simple, albeit non-trivial, example we consider, the same print job can be realized in different ways. For example, it can be done in one run or be split into several pieces, but in the latter case, some initialization work may need to be redone on the printer side for each of the pieces. In general, it may not be obvious to determine *a priori* which is the optimal way to achieve some initial goal. Instead, the user would like to be able to navigate through the space of alternatives, get a synthetic view of the scheduling possibilities in each case, and make informed decisions.

The Zippin system aims to provide such an environment. Process grammars [9] and their underlying constraints system allow a flexible definition of the navigation space. Software agents, called here robots, are in charge of managing the constraints supporting the process grammars. Robots are simple, autonomous agents [19], which cooperate without explicit communication (see [1, 8] for seminal work). Solving particular problems through a set of coordinated agents is an old technique to achieve modularity and robustness in a software system.

Process grammar rules decompose goals (recursively) into subgoals and, finally, into tasks, and introduce constraints between them. They describe a space of alternatives which can be explored through different views that give synthetic information about different aspects. The user can interact with each view, adding new constraints or removing previous ones. An interaction with a view is propagated to the underlying environment, and reflected on to the other views. Most of the interactions correspond to modifications of the constraints store and in agents' actions. Agents access the constraints store and perform the necessary actions for propagating and satisfying,

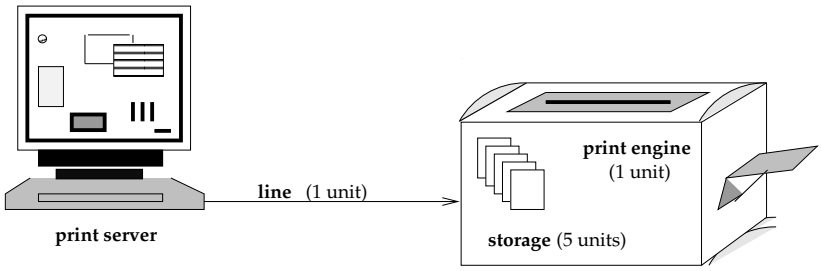

Figure 1: Sample configuration of the print task management problem.

completely or partially, the constraints. Also, agents can be activated or deactivated on demand. The agents do not communicate directly with each other, but try to fulfill given goals that are hardwired into their behavior. The effects of their actions are on the other hand propagated into the environment, and then to the user. The changes to the environment affect in turn the agents' behavior. Thus the model implements, in the form of interacting software agents, a simple type of *stigmergic* behavior. *Stigmergy*, as defined by the entomologist P. P. Grassé in 1959, refers to the actions of one agent being influenced by the effects of prior actions by the other agents. Coordinated behaviors without prior planning via communication are both common and effective in multi-agent systems (e.g. for coordinating autonomous parts of a robot [11], or individual calendars in a meeting system [5]). "Such coordination results from stigmergic sampling of the environment and responding to it" [7] (see also [3]). On the contrary, direct communication between agents is central to frameworks based on negotiation between agents (e.g. the Contract Net Protocol [17, 16] or Distributed Constraint Satisfaction Algorithms [20, 10, 6]). What is specific about our model is the combination of stigmergy with *teleology* — the view that processes are determined by their final purposes — and the use of *constraint solving* as a single computational model both for teleological and for stigmergic behavior. In fact, constraints account for goals that agents strive to satisfy, as well as for effects that are propagated into the environment as a consequence of agents' behavior.

Throughout the paper we illustrate the features of our system by a non-trivial example, involving the management of complex print tasks in a print-shop where sophisticated printing machinery is made use of in the context of a modern client-server IT architecture. Also, we illustrate how our agent model could be extended introducing agents whose actions, if desired, may fulfill specific aspects of scheduling (e.g. allocation of certain kind of resources) currently achieved by the user interacting with the graphical interface. The user could then make his(er) informed decision or ask for a computer-supported scheduling.

Organization of the paper: In Section 2, we introduce the simple print tasks management problem we are interested in. In Section 3 we formalize that problem using a grammar-based process description language. Section 4 introduces the simulation/scheduling tool Zippin, based on this process description language. In Section 5, we show an application of Zippin to our print tasks management problem. In Section 6, we go into more details on the possibilities for computer-supported scheduling. Finally, Section 7 concludes the paper.

2 A Print Tasks Management Problem

We consider a print server machine driving one or more printers connected to the server by lines. Print requests are satisfied by the server loading the corresponding print tasks into the printers using the lines. Every printer contains a certain amount of memory allowing bufferization of the data to be printed, and a printing device. Thus, there are three kinds of bounded resources to consider: the lines, the storage (in the printers) and the print engines. Figure 1 depicts a simple configuration of this kind: one printer connected to the print server by one line. The printer has one print engine and five units of storage.

We assume that the server may receive two types of print requests: *simple* and *complex*. A simple print request consists of printing one piece of data (e.g. a file). It is first loaded in the printer buffer, using the line and a certain amount of storage, then printed using the same storage and the print engine. A complex print request, on the other hand, consists of printing a number of data sets which share a large part in common, typically letters which share a common format and differ only in a few fields (address, name of recipient, order details, etc.). In this case, the form defining the common part shared by all the letters must be loaded into the printer buffer, using the line and some storage, then the individual data sets of the letters are loaded, using the line and some more storage, and printed sequentially together with the form, using the print engine.

Furthermore, it is possible and sometimes desirable to interrupt a complex request and remove the form from the buffer, to free some storage and perform some other task. When the complex request is resumed, the form must be loaded again in the buffer beforehand.

Conflicts between tasks arise from concurrent accesses to the bounded resources (the line, the storage and the print engine) hence the need to organize and schedule the accesses to these resources. We do not try to optimize any cost function (e.g. end times), but simply to respect some fixed deadlines.

Consider now the following scenario. A complex request r_1 arrives at the server with the following characteristics: the form uses three units of storage and each individual data set uses one more unit (hence, at least four units of storage are used altogether for each data set). The server also receives an additional simple print request r_2 which requires three units of storage, but is constrained by a fixed deadline. We assume that, were r_2 processed immediately, it could be finished on time, before its deadline. There are then three possibilities:

1. Request r_1 is not currently being processed: it can be delayed, thus allowing request r_2 to be scheduled immediately;

2. Request r_1 is currently being processed, but is expected to finish soon enough for request r_2 to be processed afterwards on time: r_2 is scheduled immediately after r_1.

3. Request r_1 is currently being processed and is expected to finish too late for the deadline of r_2 to be respected: r_1 must be interrupted, r_2 must be scheduled immediately and r_1 must be resumed after r_2.

In other words, the problem is not only to determine when to schedule r_1 and r_2, but also to determine whether or not, and, in the affirmative, how, to split the complex request r_1. The two problems are of course related. Notice that when more tasks and deadlines are involved, a complex task may have to be split more than once. A complex task represents a goal which can be reached by different means (here, different splits) and scheduling should take into account all the alternatives. In the following, we introduce a process description language capable of specifying such alternative ways of achieving goals.

3 Formalization of the Print Tasks Management Problem

3.1 The Process Description Language

The formalism we use to represent our sample print tasks management problem is based on Generalized Process Structure Grammars (GPSG) [9]. According to this approach, a work process is described by a set of grammar rules whose symbols represent goals. A goal may be an occurrence of a simple task (terminal symbol of the grammar) or else be decomposed into further subgoals by one or more rules. The syntax of a GPSG rule is

<*rule*>	=	<*goal*> '`->`' <*goals*> '`::`' <*constraints*>
<*goals*>	=	<*goal*>
	|	<*goal*> <*goals*>

A goal is represented as a *feature term*, i.e. a structure defining a set of attribute/value pairs. The left-hand side of a GPSG rule is a feature term that defines the matching condition for the application of the rule. The right-hand side comprises a set of feature terms, specifying the subgoals of the matched goal, and a set of constraints. Each constraint expresses a dependency among the features of the goals of the head or the body of the rule.

Every task has several predefined features. The `begin`, `end` and `duration` features characterize, respectively, the start date, the end date and the duration of the task, with the implicit constraint that the duration is equal to the end date minus the start date. The `role` feature characterizes the resources required by the task, for which it may compete with other tasks. Goals also have `begin` and `end` features, obtained, respectively, as the earliest start date and the latest end date of their subgoals.

3.2 The Sample Grammar

Figure 2 defines the complete GPSG-grammar of our print tasks management problem. Rule `(1)` introduces the main goal (`main`) which breaks down into two sub-goals, `formPrint` and `simplePrint`, corresponding, respectively, to the print requests r_1 (complex) and r_2 (simple). `formPrint` represents the printing of several letters sharing a common form. It is characterized by the feature `num` which holds the number of letters to print. `simplePrint` represents the printing of a single file.

Rule `(2.1)` specifies how to execute a `formPrint` goal in one run, i.e. when feature `status` is set to `all`. It breaks down into a `load` task, loading the common letter form in the buffer, the goal `multiPrint` for printing `N` times a single instance of the letter with its form, and the `formKeep` which maintains the form in the buffer. A number of

```
(1)   |task = main| --> |task = formPrint| |task     = simplePrint         |
      |           |     |                | |duration = 2                 |
      |           |     |                | |role     = line,mem,mem,mem,eng|


(2.1) |task   = formPrint| --> |task     = load|  |task = multiPrint|
      |status = all  |     |duration = 1   |  |num  = N         |
      |num    = N        |     |role     = line|  |                 |

                               |task = formKeep   |  ::
                               |role = mem,mem,mem|
                               |                  |

                        { formKeep starts_with load     }
           constraint = { load precedes multiPrint      }
                        { formKeep ends_with multiPrint }


(2.2) |task   = formPrint| --> |task = formPrint|  |task = formPrint|  ::
      |status = split    |     |num  = N1       |  |num  = N2       |
      |num    = N        |     |                |  |                |
      |amount = N1       |     |                |  |                |

           constraint = { N1 + N1 = N, N1 > 0, N2 > 0}


(3.1) |task = multiPrint| --> |task     = print       |
      |num  = 1         |     |duration = 1           |
      |                 |     |role     = line,mem,eng|


(3.2) |task = multiPrint| --> |task     = print       | |task = multiPrint| ::
      |num  = N         |     |duration = 1           | |num  = N1        |
      |                 |     |role     = line,mem,eng| |                 |

           constraint = { N1 = N - 1 }
```

Figure 2: The print tasks grammar.

constraints are imposed between the start dates and end dates of the different subgoals. The three constraints `load precedes multiPrint`, `formKeep starts_with load` and `formKeep ends_with multiPrint` ensure that printing cannot start before the form has been loaded in the buffer (i.e. `multiPrint` will not start before `load` has ended) and that the storage used by the form is locked throughout the whole operation (i.e. `formKeep` will cover both `load` and `multiPrint`.

As an alternative, in rule `(2.2)`, `formPrint` is split into two `formPrint` subgoals. `amount` holds the number of letter instances printed in the first subgoal. The constraint `N = N1 + N2` ensures that all the letters requested in the head goal are actually printed. Constraints `N1 > 0` and `N2 > 0` make sure that the split is not fake.

Finally, in rules `(3.1)` and `(3.2)`, the `multiPrint` goal is decomposed into a set of `N print` tasks, for printing `N` instances of a letter with a common form already loaded in the buffer.

Tasks of category `print`, `load` and `simplePrint` use some resources: the line, the storage, and the print engine. They are represented by the `line`, `mem`, and `eng` values of the `role` feature.

4 Zippin

The GPSG formalism is used in the Zippin [21] system to define a space of possible processes which the user may explore for simulation and scheduling purposes. The user proposes an initial goal and the system decomposes it according to the rules of the grammar. When several alternatives occur, the system suspends its decomposition process and waits for input from the user, who can resume the simulation by specifying new constraints on feature values of tasks or subtasks. The user can dynamically modify constraints and examine the resulting behaviour. Hence, the simulated process is represented as a net of tasks and subtasks, with dependencies among them.

Zippin offers several ways to specify dependencies between tasks. Direct dependencies, expressed by the grammar rules, result from the structure of the goals: it is assumed that a composite goal chronologically covers all its subgoals. In addition, explicit precedence constraints can be set between the different subgoals, creating further

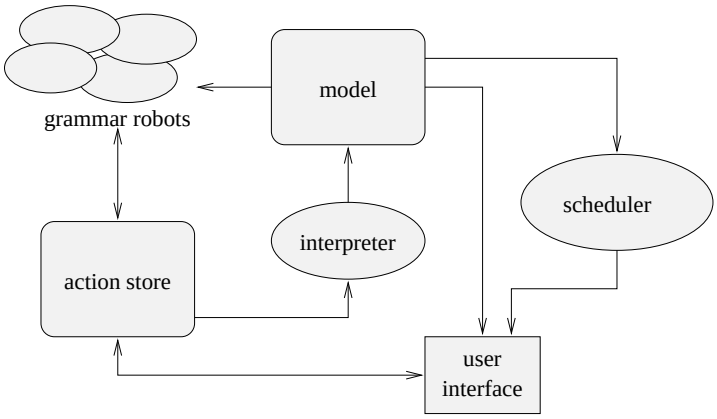

Figure 3: The Zippin architecture.

temporal dependencies. Indirect dependencies result from the competition among tasks for bounded resources: if two tasks require a common unique resource, they cannot overlap in time. The user may request Zippin to look for a solution for the allocation of the resources.

4.1 Architecture

In Zippin, the representation of a process has two different forms: an "implicit" form, the *action store*, and an "explicit" form, the *model*.

The action store is a set of actions which have to be applied for building the simulated process. An action may correspond, for example, to applying a grammar rule to a goal, or to imposing a constraint on one of the features of a task. All modifications to the simulation are implemented by adding or removing actions to/from the action store.

The model is a goal-subgoals tree of feature terms representing the current decomposition of the initial goal, together with the constraints between these feature terms. This structure is a direct representation of the simulated process.

The *interpreter* transforms the implicit form (action store) into the explicit form (model) by interpreting the actions in the store. The user inspects the model and accesses the action store through a graphical interface. The graphical interface visualizes the current simulation by several synthetic "views" that allow the user to dynamically change the simulation and examine the results. The user interacts with the views: she/he can add new constraints, relax or remove old constraints. The user interface maps user interactions into modifications of the store. In particular, the user inserts the actions describing the process grammar.

Software agents, that we call here *grammar robots*, inspect the current model and access and modify the store in order to apply the grammar. Every robot has its own domain of knowledge, and goals it tries to achieve. As envisaged in [4], the robots interface directly to the world through perception and action rather than interacting with each other: the robots' world is the model, which they all share. The robots update this model to achieve robot-local improvements and react to changes of the model by other robots, or the user. The user interface can be seen as an additional robot, controlled by the user, which translates the user interactions into store modifications.

Robots are not *benevolent* [14] and may have conflicting goals. We use a strategy based on priorities to achieve system-global goals. When a robot accesses the store, it may or may not modify it. If a robot does not modify the store, the robot with the next lower priority is activated. If a robot modifies the store, the model is updated and then the robot with highest priority is activated. The user interface robot has the lowest priority and blocks until a user interaction occurs.

The *rule_selection* robot searches for a rule to be applied to a given goal. Actions for creating the subgoals and propagating the constraints defined in a rule are managed by the *generate_tasks* and the *propagate_constraints* robots. The *check_match* robot detects situations where the conditions which allowed the application of a rule are not satisfied anymore.

The main advantage of this approach is the modularity. Each robot may be switched on/off individually, taking/giving back the control to the user. For example, switching off the *generate_tasks* robot, the user has full control over which tasks should appear in the simulation and which tasks should be omitted, while still using automatic rule selection.

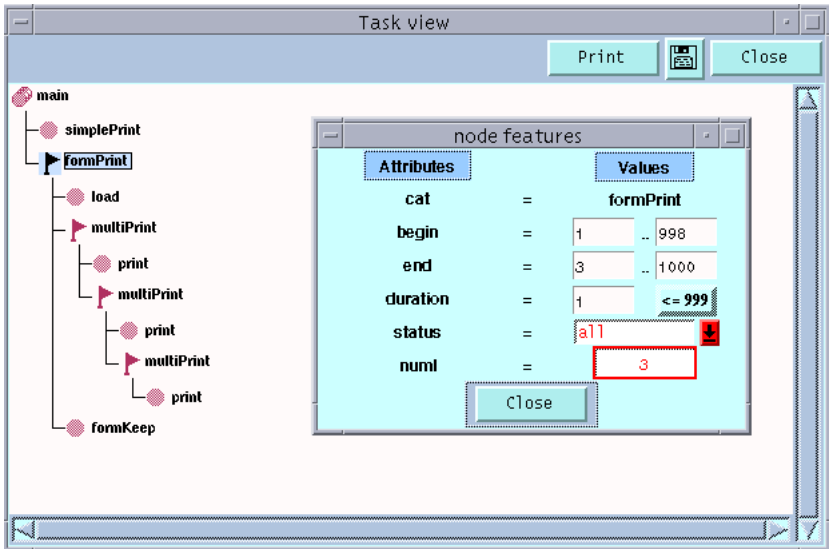

Figure 4: The task structure of a `formPrint` task.

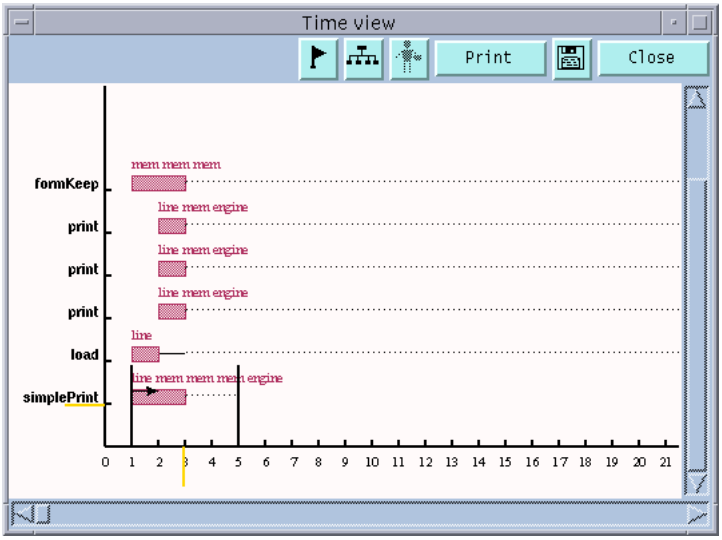

Figure 5: The time constraints of the tasks.

Moreover, it is possible to incrementally extend the system by adding new domains and goals, and by improving the strategies used by single robots. In Section 6, we consider other possible cases in which automatic modifications of the action store could be performed by robots. This architecture is depicted in Figure 3.

4.2 Synthetic Views

In the following, we shall concentrate on two synthetic views of the user interface, the so-called *task view*, and *time view*.

- The task view shows the goal/sub-goals dependencies between goals and tasks in a hierarchical structure. Each element in the tree represents a goal or a task. A task is represented as a circle labelled with the name of the task. A goal is represented as a square containing a question mark if it marks a *decision point* or by a flag if it marks an instantiated decision point. Otherwise, a goal is represented with overlapping circles. A goal marks a *decision point* if its decomposition is suspended because several alternatives exist for its decomposition. The features of a goal/task are displayed when the corresponding node in the tree is selected. The user can insert constraints on the current state of the goal/task. Once a node is selected, the user can

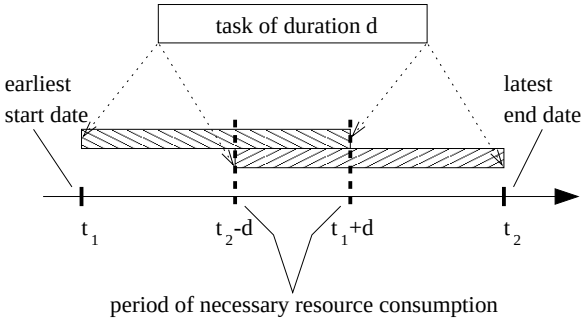

Figure 6: Determination of a period where a resource is *necessarily* consumed.

choose an entry of the feature list and specify a value for that feature. The system will either perform the assignment, if consistent with the current status of the simulated process, or will refuse to perform it, if the global constraints are violated. Figure 4 is an example of interaction through the task view.

- The time view visually depicts the current constraints (explicitly inserted and/or resulting from the propagation) on the start dates and end dates of the tasks in the process. The timing constraints on a given task are visualized as follows (see Figure 5). Each task is represented as a box the length of which is proportional to the duration of the task. The left side of the box is the earliest possible start time of the task. Its color is grey when the start time is not fully known, but only bounded. The user can enforce or relax constraints by moving the box for the task along the horizontal axis (thus changing the earliest start date).

4.3 Allocation of the Resources: the Scheduler

In Zippin it is possible, at any point, to request the system to find a solution for the allocation of the resources to the tasks in the current state of the simulated process. We use a technique similar to the micro-opportunistic scheduling of [15].

Once given a set of resource bounds, the scheduler searches for an allocation of the resources, respecting the given bounds and the other constraints, while trying to minimize the execution time for the whole process. Each time a solution is found, it is displayed on the different views and the user may either accept it or request further search for another solution. The scheduler maintains, for each resource type, a temporal map, called the "usage profile", which gives, at any point of time, the number of units of that resource which will *necessarily* be consumed at that time, given the timing constraints of the tasks. Thus, if, for example, a task of duration d is constrained to start after t_1 and to end before t_2, then it is necessary that the resources specified in the role of that task will be consumed during the period between $t_2 - d$ and $t_1 + d$ (see Figure 6).

When the boundary of a resource is reached at some date d on its usage profile, then all the tasks which consume the resource, except those which have been counted as necessarily consuming the resource at date d on the usage profile, are constrained to either *end before* d or *start after* d (choice points are created here, we do not attempt to propagate disjunctive constraints [2]).

5 Print Tasks Management in Zippin

Figure 4 depicts the task view of a Zippin session, after starting the simulation with the print tasks grammar given in Section 3 and with the initial goal `main`. Furthermore, we have selected the `formPrint` node, set the `num` feature to 3 and the `status` feature to `all`. Three `print` tasks have been generated.

According to the scenario in Section 2 we have three possibilities for scheduling the goal `formPrint`, that corresponds to request r_1, and the task `simplePrint`, that corresponds to request r_2:

1. `formPrint` is not split and is performed after `simplePrint`;
2. `formPrint` is not split and is performed before `simplePrint`;
3. `formPrint` is split in two `formPrint` subtasks performed, respectively, before and after `simplePrint`.

Suppose now that we are in scenario 1., modelling the case where the `formPrint` task has not yet started when the server receives the request for the task `simplePrint`. Moreover, suppose that the deadline for the `simplePrint`

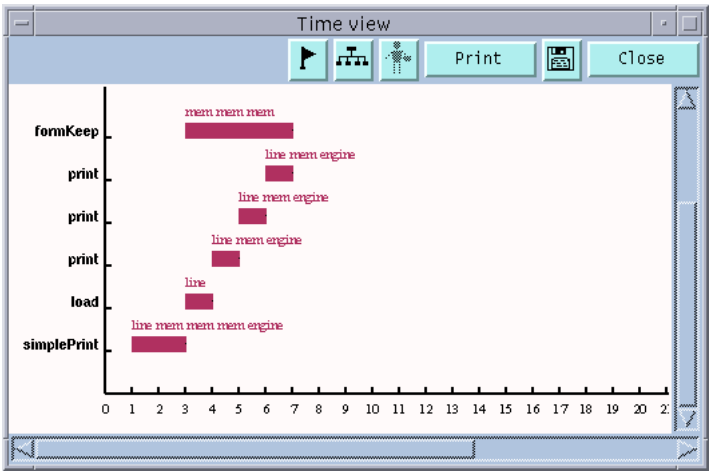

Figure 7: The simple print request is accomplished before the complex print request.

is 5. Figure 5 shows the feasible schedule of the tasks, after the deadline has been set to 5 for `simplePrint`. The deadline implies that the `simplePrint` must be perfomed inside the region delimited by the two vertical lines.

Figure 7 shows a solution for the allocation of the resources found by the scheduler, given the set of available resources: the simple print request is accomplished before the complex print request.

Consider now the scenario in 2., which corresponds to the case where the `formPrint` task has already started, and in particular suppose that, when the server receives the request for the task `simplePrint`, the `load` task and the first `print` task have already been performed. This means that, for these two tasks, we already know the begin and the end times (acknowledged by the change of color of these tasks on Figure 8). The `load` task has begun at 1 and finished at 2, and the first `print` task has begun at 2 and finished at 3. Moreover, suppose that the deadline for the `simplePrint` is 13.

Figure 9 shows a solution for the allocation of the resources found by the scheduler: the complex print request is accomplished before the simple print request because this does not violate the deadline for the simple print task.

In scenario 3., we suppose that, as in scenario 2., the `formPrint` goal has already started when the server receives the request for the task `simplePrint`: the `load` task has begun at 1 and finished at 2, and the first `print` task has begun at 2 and finished at 3, but now the deadline for the `simplePrint` is 5. In this case, the scheduler tells us that it cannot find a solution. As a matter of fact, the deadline on the simple print request cannot be respected without splitting the complex print request into portions. We then need to insert a new `load` task for the last two `print` tasks. To represent this, we dynamically change the rule for the complex print request, that is, instead of rule (2.1) we have to apply rule (2.2). To obtain this, we dynamically change the `status` value for the `formPrint` goal from `all` to `split`. Zippin updates the simulation according to this new value applying rule (2.2) to the `formPrint` goal. The `formPrint` goal is composed now by two `formPrint` subgoals. Now we distribute the three `print` tasks of the `formPrint` goal setting `num` of the first `formPrint` to 1 and of the second one to 2, and the `status` to `all` for both. Moreover, we know that a `load` task and a `print` task have already been performed, then we set the end time of the `load` task to 2 and the end time of the `print` task to 3. Figure 10 shows the changes in the simulated process.

If we now ask the scheduler to find a solution for allocating the resources, we get the solution of Figure 11. The complex print request has been split into two portions, with the simple print request being performed in between.

6 Supporting Simulation and Scheduling with Robots

In the previous section, we have shown how the user can guide the simulation in order to obtain a feasible schedule for a given problem. Moreover, we have seen that specialized robots can perform the application of the process grammar rules on behalf of the user. We are experimenting this approach also for automating specific aspects of the simulation, e.g. allocating certain kinds of resources. In this section we briefly illustrate our strategy.

One possibility is to introduce a robot in charge of detecting and handling cases where a certain task of category `formPrint` cannot be split without violating the constraint `formKeep ends_with multiPrint`. Let's call this robot the *form_robot*. Its behavior is based on the following heuristic. It reacts to the fact that the scheduler cannot produce a solution because of one specific constraint (`formKeep ends_with multiPrint`). If, by relaxing this

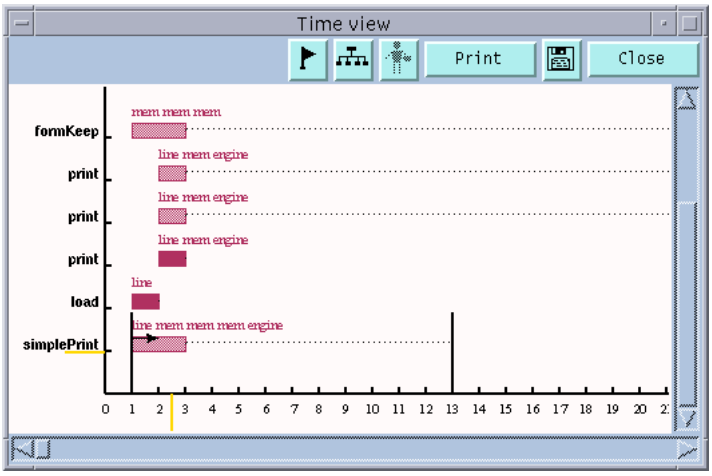

Figure 8: The `load` and the first `print` tasks have already been performed.

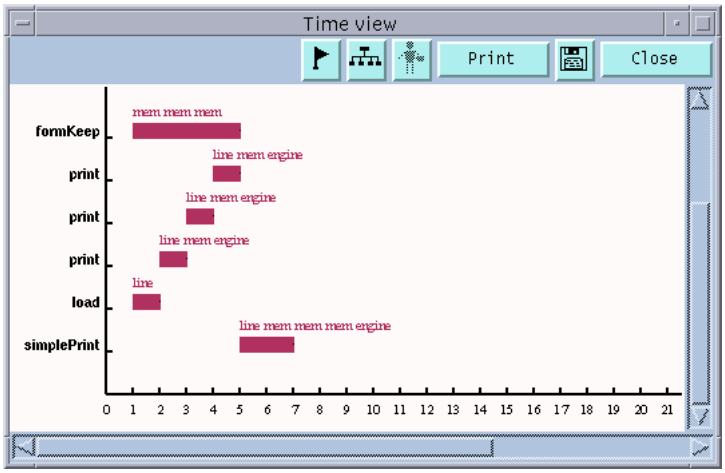

Figure 9: The complex print request is accomplished before the simple print request.

constraint, a solution is found, it attempts to maximize the length of the `formKeep` subtask and counts the number of `print` subtasks covered by the `formKeep` subtask in that case. Let M be this number. The *form_robot* then splits the `formPrint` task by setting its `status` feature to `split` and its `amount` feature to M. Once the model is updated, the `formPrint` task will be split into one part with M `print` subtask and another part with the rest of `print` subtasks. Each part has its own `formKeep` subtask. The scheduler will then be able to schedule all tasks satisying all the constraints.

On the other side, it could be useful to have the opposite capability. Assume that, for some reasons, after splitting the `formPrint` task and scheduling the `simplePrint` task between the two parts, the `simplePrint` task is cancelled before it has begun to be executed. Now, the system will push the second part of the `formPrint` task towards its first part, but then the form will be loaded twice, although this is not necessary any more. By adding a robot which joins `formPrint` tasks whenever possible, we can handle this situation. Let's call this robot *reuse_robot* since it makes the `print` tasks of the second part able to reuse the loaded form instead of loading the form again. The robot looks for split of `formPrint` tasks where the memory resources between the end of the first part and the beginning of the second part are sufficient to keep the form. If it detects such a case it resets the `status` feature of the `formPrint` task to `all` (backtrack), thus merging the two parts and omitting the unnecessary `load` subtask.

We want to stress several properties of our approach. First, every robot has its own domain of knowledge which it observes and uses in the world. Second, every robot has its own goal or goals it tries to achieve in the world. For example, the form robot looks at covering constraints of specific tasks. The goals of different robots may conflict, thus a mechanism of priorities among them has been adopted. Altogether, every robot abstracts the

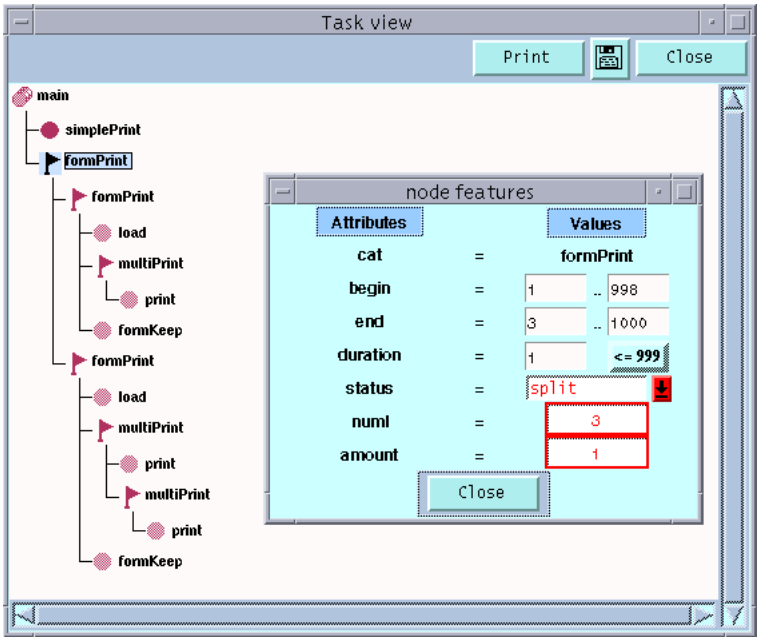

Figure 10: New task-subtask structure.

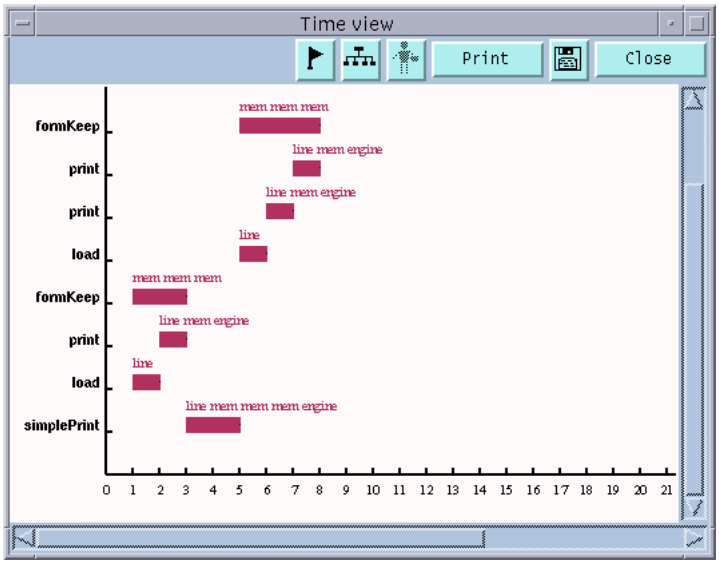

Figure 11: Simple print task scheduled between two data settings of the complex print request.

full information available in the world to the part belonging to its domain and using only this part as a basis for its decisions. This helps in defining simple strategies for each single robot, even if their world is complex. However, the fact that every robot has its own domain does not prevent their collaboration. The collaboration is implicit. The robots do not communicate directly, they do not even use any knowledge about the existence of the other robots. And there is no global coordinator which leads them. A robot does not try to anticipate actions of other robots, it simply observes the consequences of their actions in the world afterwards. Robots can be removed or added, without disturbing the work of other robots. The only effect will be that the behavior emerging from the collective effort of the robots will be different. On the other hand, this makes it possible to shape the overall behavior by incrementally adding robots.

7 Conclusions

We showed how to handle a non-trivial print tasks management problem within Zippin. We presented the formalization of the problem through a grammar-based task description language. Synthetic views of a typical scheduling example were illustrated throughout the paper.

We introduced autonomous computing agents (called here robots) supporting the user in the application of the process grammar. These concurrent robots cooperate without explicit communication. We described then some extensions, that we are exploring, to enrich the system with new robots which could support the user during the cumbersome scheduling process. Moreover, we discussed some properties of our approach.

Our focus is on supporting decision makers using constraints and agents based scheduling techniques, with a particular stress on the user interface (essential in this context) which allows the control of the constraint resolution steps performed by the agents (including the user him/herself). In this sense, we are close to "mixed-initialive" problem solving systems, such as IC-6 [13] or Inova [18]. The IC-6 Intelligent scheduling project at University of Alberta has similar objectives, but does not rely on agent technology. It exploits intelligent backtracking and belief revision techniques for the control of the scheduling process. The Inova project of the AI institute in Edinburgh uses control agents with constraints, but is more oriented towards planning and agenda conflict resolution.

The system is up and running. It has been implemented in Eclipse [12], a Prolog environment incorporating powerful constraint solving facilities. The graphical user interfaces have been developed in Tcl/Tk for which Eclipse provides interfaces. For more information concerning the Zippin project, visit our Web-site at `http://www.xrce.xerox.com`.

References

[1] R. C. Arkin. Cooperation without communication: Multiagent schema-based robot navigation. *J. Robotic Systems*, 9(3):351–364, April 1992.

[2] P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proc. of IJCAI'95*, Montreal, Quebec, Canada, 1995.

[3] R. Beckers, O. E. Holland, and J.-L. Deneubourg. From local actions to global tasks: Stigmergy in collective robotics. In R. Brooks and P. Maes, editors, *Proc. 4th Int. Workshop on the Sythesis and Simulation of Living Systems (Artificial Life IV)*, pages 181–189. Cambridge, MA: MIT Press, July 1994.

[4] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[5] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12:251–284, 1994.

[6] S. Conry, K. Kuwabara, V. Lesser, and R. Meyer. Multistage negotiation for distributed constraint satisfaction. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1462–1477, 1991.

[7] S. Franklin. Coordination without communication. *Available at http://www.msci.memphis.edu/~franklin/coord.html*, February 1996.

[8] M. R. Genesereth, M. L. Ginsberg, and J. S. Rosenschein. Cooperation without communication. In *Proc. 5th Nat. Conf. of Artificial Intelligence*, Philadelphia, PA, 1986.

[9] N. Glance, D. Pagani, and R. Pareschi. Generalized process structure grammars (gpsg) for flexible representations of work. In *Proc. 7th Europ. Conf. on Computer-Supported Cooperative Work*, Boston, MA, November 1996.

[10] J. Liu and K. Sycara. Emergent constraint satisfaction through multi-agent coordinated interaction. In *Proc. of the 5th European Workshop on Modelling Autonomous Agents in a Multi-Agend World (MAAMAW'93)*, pages 107–121, Neuchâtel, Switzerland, 1993.

[11] P. Maes and R. A. Brooks. Learning to coordinate behaviors. In *Proc. 8th Nat. Conf. on Artificial Intelligence*, pages 796–802, Menlo Park, CA, 1990. AAAI Press.

[12] M. et als Meier. Eclipse user manual. http://www.ecrc.de/eclipse/eclipse.html.

[13] University of Alberta. Iris project ic-6: Intelligent scheduling. http://www.cs.ualberta.ca/ ai/IC-6.

[14] J. S. Rosenschein and M. R. Genesereth. Deals among rational agents. In *Proc. 9th Int. Joint Conf. on Artificial Intelligence*, pages 91–99, Los Angeles, CA, August 1985. also in: A. H. Bond and L. Gasser (eds.): Readings in Distributed Artificial Intelligence, pp. 227–234, 1988, Los Altos, CA: Morgan Kaufmann.

[15] N. Sadeh. Micro-opportunistic scheduling: The micro-boss factory scheduler. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*, San Francisco, Ca, U.S.A., 1994. Morgan Kaufmann Publishers.

[16] T. Sandholm and V. Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *Proc. of the 1st Int'l Conference on Multi-Agent Systems*, pages 328–335, San Francisco, Ca, U.S.A., 1995. MIT Press.

[17] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C–29(12):1104–1113, December 1980.

[18] Austin Tate. Representing plans as a set of constraints - the ¡I-N-OVA¿ model. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 221–228. AAAI Press, 1996.

[19] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[20] M. Yokoo. Asynchronous weak commitment search for solving distributed constraint satisfaction problems. In *Proc. of the 1st Int'l Conference on Principle and Practice of Constraint Programming, CP'95*, pages 88–102, Cassis, France, 1995.

[21] URL. Zippin: http://www.xrce.xerox.com/research/ct/prototypes/zippin/home.html.