Interactive Constraints for Plan Construction and Execution

R.Barruffi[(1)], M.Milano[(2)], P.Torroni[(1)]

[(1)] DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{rbarruffi, ptorroni}@deis.unibo.it

[(2)] Dipartimento di Ingegneria, University of Ferrara
Via Saragat 1, 41100 Ferrara, Italy
mmilano@deis.unibo.it

Abstract

Traditional planners need a complete representation of the initial state of the environment they are acting on. In complex and dynamic settings, however, the information describing the state of the environment can be too big to be efficiently stored and updated, and changes frequently. Thus, the representation the planner has of the real world it is acting on can become obsolete during or after plan construction. The corresponding off-line plan execution can fail. We propose a solution where a planner and an executing component interact in order to face dynamic changes. They are able to retrieve knowledge on the environment only when needed, thus delaying as much as possible data acquisition and minimizing, at the same time, data to be managed. They exploit the Interactive Constraint Satisfaction framework [7]. This framework represents an extension of the constraint satisfaction framework based on Interactive Constraints (ICs) which are declarative relations stated among variables whose domain is partially (or fully) unknown. In presence of incomplete knowledge, ICs act both as classical constraints, in order to prune inconsistent information, and as sensing modules to the environment, in order to retrieve necessary (and consistent) information. We exploit ICs in planning both for acquiring knowledge about the environment during plan construction and for testing the (possibly changed) environment during plan execution. In particular, the plan is refined at execution time by retrieving new consistent values when those available are no longer consistent. In this setting, both planning and execution can be seen as search processes in the space of partial plans. Since execution produces side effects on the environment, a backup and recovery mechanism should be provided in order to allow backtracking at execution time. A case study in the field of computer system management shows the advantages of the proposed solution.

1 Introduction

Traditional generative planners [11] assume that:

- correct and complete knowledge about the application environment is given;
- the plan executor is the only cause of change of the environment;
- action execution is atomic (i.e., it is non interruptible);
- the effects of executing an action are a deterministic function of the action and the current state of the world.

A consequence of these assumptions is that often a planner represents an *off-line* system unable to interact with the environment it is acting on.

In dynamic and complex environments, we need to cope with three major problems: (i) the environment where the planner acts can change before or during the execution of the plan; (ii) the action can have unpredictable effects; (iii) the amount of data to be considered can be huge and data retrieval in some cases can be computationally expensive.

As a consequence, we need to provide:

- a sensing mechanism devoted to retrieve the current state of the environment during plan construction. Sensing should be aimed at minimizing the amount of retrieved data;
- a sensing mechanism for testing the environment during plan execution;
- a procedure for plan refinement if the environment has changed and the execution of the current plan fails.

We face these problems by using a general constraint based framework, called Interactive Constraint Satisfaction Problem (ICSP), proposed in [7]. *Interactive Constraints* (hereinafter referred to as ICs) are declarative relations among variables whose domain (i.e., the set of values the variable can assume) is possibly partially or fully unknown. Their operational behaviour extends standard constraint propagation with a data acquisition mechanism devoted to retrieving values for variable domains.

Note that if proper sensors to the environment are available, only consistent information is retrieved thus minimizing the amount of knowledge to be managed by the planner. As a simple example in the field of computer management, consider a variable P representing a printer subject to the constraint imposing that it should be connected to a certain machine, i.e., `connected(P,machine1)`. The acquisition of domain values for P can be guided by the constraint and limited to a particular machine, i.e., `machine1`.

For this purpose, action preconditions and final goal conjuncts are considered as ICs [1]. Variables appearing in ICs are domain variables and represent resources in the system, and domain values represent alternative resources. Each time an open condition needs to be achieved during plan construction, the corresponding IC is triggered. If variables involved in the open condition have a corresponding known domain, standard constraint propagation is performed aimed at removing combinations of assignments which cannot appear in any consistent solution. Otherwise, data acquisition is performed in order to retrieve (possibly) consistent values for variable domains.

During plan construction, all significant data for the planner is retrieved to create variable domains. Acting in dynamic environments, resources available at plan construction time can be no longer available at plan execution time or new resources can be at disposal.

We exploit the Interactive CS framework to cope with dynamic environments and non deterministic action effects. Basically, we argue that the IC propagation/acquisition mechanism can be used not only for gathering information, but also for verifying preconditions before action execution. In this way, we can monitor the system behaviour and check if it corresponds to the one expected by the planner.

Preconditions are tested during plan execution on available data. If the test succeeds the action is executed. On the contrary, if already available data are no longer consistent, they are pruned from variable domains. If none of the domain values is consistent, the plan is refined by either retrieving new consistent values if any, or backtracking if a dead end occurs[1]. Since execution produces side effects on the environment, a backup and recovery mechanism should be provided in order to allow backtracking at execution time.

The advantages of this approach are several:

- the planner should not load all the information about the environment before starting the plan construction;
- only useful data is retrieved thus minimizing (possibly computationally expensive) information extraction operations;
- the information gathering process is transparent to the planner which relies on a constraint solver without taking into account declarative sensing actions;
- dynamic changes are handled by testing ICs during plan execution;
- new data is processed without starting the Constraint Satisfaction process from scratch, but minimizing plan changes.

The main point of this paper is to describe how CS techniques can deal with the problem of incomplete and changing knowledge in planning. In section 3, we face the problem of incomplete knowledge, while in section 4, we describe how the ICs-based approach can be used also for coping with dynamic changes.

2 Preliminaries

In this section, we provide some preliminaries on Interactive Constraint Satisfaction Problem framework proposed in [7]. Interactive constraints are relations among variables ranging on partially or completely unknown domains. As in traditional CSPs, a solution to

[1] In this setting, plan execution can be seen as a search process in the space of partial plans.

the ICSP is found when all the variables are instantiated consistently with constraints. A formal definition of the ICSP framework follows.

Definition 2.1 *An* ICSP *is a tuple* (V, D, C) *where* V *is a finite set of variables* $X_1, \ldots, X_n$*,* D *is set of interactive domains* $D_1, \ldots, D_n$ *and* C *is a set of interactive constraints. An* interactive domain D_i *of a variable* X_i *is the set of all possible values that can be assigned to that variable.* $\forall i \; D_i = [Known_i \cup UnKnown_i]$ *where* $Known_i$ *represents the set of known values for variable* X_i*, and* $UnKnown_i$ *is a domain variable itself representing (intensional) information which is not yet available for variable* X_i*. Both* $Known_i$ *and* $UnKnown_i$ *can possibly be empty*[2]*.* $\forall i \; Known_i \cap UnKnown_i = \emptyset$*. A (binary)* interactive constraint IC *on variables* X_i *and* X_j*, defines a possibly partially known subset of the cartesian product of variable interactive domains* $D_i \times D_j$ *representing sets of possible assignments of values to variables. The syntax of an interactive constraint is* $IC(c(X_i, X_j))$*.*

Given an interactive constraint $IC(c(X_i, X_j))$, its operational behaviour is the following:

1. **If** both variables are associated to a partially or completely unknown domain, the constraint is suspended;

2. **else**, if both variables range on a completely known domain, the constraint is propagated as in classical CSPs;

3. **else**, if one variable (say X_i) ranges on a fully known domain and the other (X_j) is associated to a fully unknown domain a knowledge acquisition step is performed; this returns either a finite set of consistent values representing the domain of X_j, or an empty set representing failure.

4. **else**, if X_i ranges on a fully known domain and X_j is associated to a partially known one, X_j domain is pruned from values non consistent with X_i. If X_j domain becomes empty a new knowledge acquisition step is performed for X_j driven by X_i.

This is a general framework which can be used in many applications. It is particularly suited for all the applications that process a large amount of constrained data provided by a lower level system, see for instance [6, 7].

3 ICs for Plan Construction

Our planner is a regressive Partial Order Planner (POP) [11] performing least commitment planning aimed at preventing premature (possibly wrong) decisions.

Constraint Satisfaction (CS) techniques can be used in planning in order to drive the planner in its commitments so as to avoid non deterministic decisions and possible consequent failures, thus reducing the problem search space. In order to exploit these techniques, there is need to map the planning problem into a CSP. In particular, we

[2] When both are empty an inconsistency arises.

focus on CS-based variable binding activity [1]: the correspondent CSP variables are variables appearing in action schemata introduced into the plan. The constraints are the usual nocodesignation ($\neq$) and codesignation ($=$) constraints imposed among the problem variables during the planning process. Variable domain values are sets of alternative resources in the domain description. Moreover, we consider action preconditions as Interactive Constraints among variables appearing in the same action instance. For example, the precondition *connected*$(Printer, Machine)$ represents a constraint linking variables $Printer$ and $Machine$ which is satisfied by any couple of values (p, m) where p represents a printer connected to machine m. Both variables can be associated with a domain of alternative values.

Thus, every time the planner selects an open condition $p(X, Y)$ (i.e., an action precondition or a final goal conjunct), the constraint solver tests if there exists at least one value for X and Y that already satisfies p in the initial state. When variables (X, Y) range on known domains, traditional constraint propagation is performed in order to prune the domains from inconsistent values. Otherwise, constraint propagation results in acquisition of domain values from the real environment.

It is worth noting that those constraints retrieve only consistent information so as to simplify the task of pruning inconsistent alternatives.

Given the example above, suppose only the $Machine$ domain is known (e.g, $Machine :: [m1, m2]$), the *connected*$(Printers, Machine)$ propagation will result in the acquisition of the $Printers$ domain, which will contain only the printers connected to $m1$ and $m2$.

If the constraint fails (i.e., a variable's domain becomes empty), it means that the corresponding precondition is not satisfied in the initial state. Thus, the planner selects an existing action, or a new one, whose effects contain a conjunct $p(X1, Y1)$ unifiable with the open condition $p(X, Y)$. Necessary codesignation constraints ($X = X1$, $Y = Y1$) are added to the constraint store.

All the problem constraints are awaked, during the planning process, when some variables get instantiated or their domain pruned, until all the involved variables are instantiated, as in classical CSPs .

The algorithm ends when all preconditions are guaranteed to be satisfied in a consistent way with all the constraints of the plan.

4 ICs for Plan Execution

The enhanced complexity of planning techniques when applied to dynamic environments is due to the facts that (*i*) typically the planner is not the only agent that causes changes on the system and (*ii*) often changes are not deterministic. This can lead to a failure of the plan execution, either because action preconditions are no longer verified at execution time, or because action effects are not those expected because of a non deterministic behaviour of the system.

In order to cope with the problem of dynamic knowledge there is need of:

- delaying as much as possible the variable binding activity since variables represent alternative resources whose state can change during or after plan construction;

- providing a run-time precondition verification mechanism which checks whether action preconditions (corresponding either to the initial state or to previously executed action effects) are still true at execution time;
- providing an executable backtracking mechanism to "retract" action effects when the real world situation after action execution does not correspond to the one expected by the planner.

As far as the first point is concerned, the planner returns to the executing component a plan where variables appearing in action preconditions are still unbound and are associated to a domain of alternative resources. With respect to Constraint Satisfaction Problem resolution, we delay the variable labelling step at execution time[3]. As a consequence, the planner does not return a single final solution, but a set of alternative plans containing the same set of actions acting on different resources.

Action preconditions are tested on the system thanks to the same ICs propagation mechanism exploited for data acquisition. When variables appearing in preconditions are already instantiated, the interaction with the underlying system is simply a test returning a boolean value. Otherwise, if variable domains contain more than one value, these values are checked for consistency at execution time. Values no longer consistent with the current state of the system are removed at execution time. Once a consistent value is found, the variable is instantiated. Note that value removal can trigger constraint propagation which, in turn, removes values from other variable domains, thus reducing the *execution search space*.

For instance, suppose that, during the planning process, a printer needs to be selected to print a job. Suppose, also, that at the end of the first phase of the planning algorithm, after constraint propagation, the printer domain still contains printer objects $p1$ and $p2$. If, during plan construction, the algorithm non deterministically selects one of the two printers, say $p1$, it might happen that, at execution time, $p1$ is no longer available (while $p2$ is still available) and the plan would fail. On the other hand, if we delay the commitment on the *printer* variable instantiation until execution, $p1$ is pruned from the variable domain, $p2$ is left and no failure occurs. Thus, the plan execution process results in a search process in order to find the proper action variable instantiation before performing them on the real world.

If a domain becomes empty, the planner has two possibilities: either fail and backtrack, or look further for a new set of consistent values. The second case implies a more complex mechanism for coping with non monotonic changes. This mechanism will be discussed in section 5.

Our repair mechanism, based on backtracking, is performed in order to select an alternative plan action instance. It also supports the cases in which the planner, through IC-driven verification, realises that the effects of the action are not those expected. As far as our application domain (computer systems management) is concerned, two different kinds of action can be recognized: *backtrackable* and *non backtrackable* actions. *Backtrackable* action effects can be retracted by a single action or a set of actions (*backtracking*

[3] In a CSP, variables get instantiated when only one value has been left in variable domains because of constraint propagation, otherwise a search step, called labelling step, is needed.

plan) dynamically computed at execution time. *Non backtrackable* action effects are unrecoverable unless one or more *backup actions* are inserted in the plan at plan construction time. We are currently enriching the action domain with *Backup* actions. The insertion of backup actions in the plan is allowed by the presence of backup preconditions in *non backtrackable* actions. For instance, an action aimed at deleting a file in the Unix operating system has as a precondition the fact that a backup file exists. Thus, the planner will automatically insert into the plan an action devoted to copying the file to be removed, simply as result of a normal open condition achievement step.

5 Non monotonic changes

Up to now, we have considered that values acquired during plan construction can be no longer available during plan execution. However, a more complex situation occurs when some new values are available during plan execution and have not been retrieved during plan construction. Standard CSPs do not cope with value insertion in variable domains since it implies reconsidering previously deleted values which can be supported by the newly inserted value. Dynamic Constraint Satisfaction (DCS) [8] has been proposed in order to deal with non monotonic changes. DCP solvers maintain proper data structures so as to tackle modifications of the constraint store.

In this perspective, also our ICSP framework copes with changes since the acquisition of new variable values can be seen as a modification of the constraint store. For dealing with non monotonic changes, we adopt a different approach with respect to DCSP: we leave variable domains open.

An *open domain*[4] is represented by a set of known values and a variable representing the unknown domain parts, i.e., potential future acquisitions. The propagation we perform is less powerful than that performed by dynamic approaches. In fact, if we consider a constraint between variables `X` and `Y`, the variable inserted in the domain of variable `X` represents a potential support for values in the domain of variable `Y`, which cannot be pruned until the domain of `X` becomes closed. However, we do not need to store additional information for restoring the constraint store consistency as done by DCS approaches.

During plan execution, if the entire set of known values (those acquired during plan construction) is deleted because of precondition verification, a new acquisition can start aimed at retrieving new consistent values and executing the plan on them. If no values are available, backtracking is performed in order to explore the execution of other branches in the search space of partial plans. If the overall process fails (and only in this case), a re-planning is performed.

[4]An open domain can be implemented by using the notion of interactive domain presented in section 2.

6 Case Study: Planning and Execution in Computer System Management

An application area of our planner-executor is the automated management of processes and services in a computer system. In fact, many of the tasks normally carried out by a system operator, such as activating daemons, resetting crashed applications and so on, are not in general easy tasks to be faced by a simple operating system procedure. They could require some data acquisition, several tests, on the fly decisions and actions, and much experience. Computer systems are highly dynamic environments. Thus, configuration, maintenance and recovery procedures may not be easily standardized.

We think that our solution can be a suitable tool for creating and executing plans in such an environment. Let us see a simple example devoted to understanding the role played by the two components: planner and executor. Let us consider a traffic monitoring application, which requires some support processes being *up and running* over a LAN. Those processes are called *triggers*, and are devoted to controlling the application by activating or deactivating other processes at run time. When a malfunction of the application is detected, the correct behaviour of all the *trigger* processes should be re-established. The planner comes into play and builds a series of alternative plans whose goal is having the application up and running again. After the planning phase, the system operator is asked to choose one among the alternative plans. Then, the executor performs all the actions of the chosen plan, until either the goal is achieved or at some step some action preconditions are not verified. In this second case, there is need of proper counteractions, possibly one more planning step, and so on. The failure of the precondition could be due to a previous action which did not provoke the desired effects, or to external actions which modified the system independently from those performed by the executor.

To complete the picture, let us describe what the application needs to run properly, and to start up again in case of malfunction. The *trigger* processes are launched by a daemon process whose name is *triggerStart*. Aim of a *trigger* process is to activate in its turn the processes which all together constitute the application and whose code location inside the system is specified in a file called *TMAtasks*. Moreover, the application can be started only if its configuration is that allowed by the *runlevel* of the system. The system runlevel can be seen as a software configuration of the system that allows only a selected group of processes to exist. As far as our application is concerned, its configuration depends on the location of the file containing the bootstrap code, *TMAboot*, which launches *triggerStart*. In the system there could be multiple copies of file *TMAboot*. In a correct configuration, this file is kept in a directory called */sbin/rcX*, where X is a number corresponding to the current runlevel of the system[5].

All the information that the planner needs in order to build a plan is acquired from the system at planning time and kept in the form of Prolog clauses. An important point is that the planner does not acquire all data describing the environment before starting the plan construction. Therefore, its initial state will contain a subset of the

[5]There are as many */sbin/rcX* directories as possible alternative system runlevels.

whole knowledge. Suppose we have a domain represented by a machine m at runlevel 3, five directories */sbin/rcX* corresponding to five runlevels, two instances of file TMAboot located in directories */sbin/rc2* and */sbin/rc5*, a process triggerStart which is currently up, and a process trigger which, being not up, causes a wrong application configuration. Let us describe the situation with a series of facts. Note that those facts are not in the knowledge base of the planner until it senses them.

```
machine(<machine> m,<address> ipaddress,<OS> hpux,<runlevel> 3,<status> on).
dir(<path> '/sbin/rc0',<rights> rw,<machine> m,<runlevel> 0).
dir(<path> '/sbin/rc1',<rights> rw,<machine> m,<runlevel> 1).
dir(<path> '/sbin/rc2',<rights> rw,<machine> m,<runlevel> 2).
dir(<path> '/sbin/rc3',<rights> rw,<machine> m,<runlevel> 3).
dir(<path> '/sbin/rc4',<rights> rw,<machine> m,<runlevel> 4).
dir(<path> '/sbin/rc5',<rights> rw,<machine> m,<runlevel> 5).
file(<name> TMAboot,<rights> rw,<path> '/sbin/rc2',<type> exe,<size> 50,<runlevel> 2).
file(<name> TMAboot,<rights> rw,<path> '/sbin/rc5',<type> exe,<size> 50,<runlevel> 5).
process(<name> triggerStart,<status> on,<process_ID> pid,<timestamp> time).
process(<name> trigger,<status> off,<process_ID> null,<timestamp> null).
```

As soon as a malfunction is detected, the planner is started. Its goal is to have the application working properly which means:

- TMAboot located in the directory corresponding to the runlevel of the system;
- both triggerStart and trigger up and running.

According to the current state of the system, the planner generates a certain number of alternative (possibly non ground) plans. Let us take a look at two of those.

- killProcess(*<proc_name>* triggerStart),
copy(*<file_name>* TMAboot, *<from_path>* X, *<to_path>* /sbin/rc3),
onTriggerStart(*<at_runlevel>* 3, *<at_path>* /sbin/rc3).
- changeRunLevel(*<machine>* m, *<from_runlevel>* 3, *<to_runlevel>* I),
reboot(*<machine>* m, *<at_runlevel>* I, *<TMAboot_path>* X).

For the sake of semplicity, in discussing the first plan execution, we consider only monotonic changes after plan construction, i.e., some resources available during plan construction are no longer available during plan execution. In discussing the second plan, instead, we consider also non monotonic changes (variable domains are left open).

The first plan says: *"kill the process* triggerStart*, then copy the file* TMAboot *from its current location to directory* /sbin/rc3 *and restart the daemon* triggerStart *at runlevel* 3*"*. Since two instances of the file TMAboot exist, variable X is associated to a domain which contains all the paths where TMAboot was found at data acquisition time: Dom(X)=[/sbin/rc2, /sbin/rc5]. During plan execution, when the action copy is performed, the planner-executor[6] checks if the domain values are still valid, selects a value in the domain, and therefore an instance of the file to copy. Suppose that in the meanwhile, directory /sbin/rc2 has been removed, while /sbin/rc5 still exists. Thus, the domain of variable X is pruned at execution time and X is labelled with a still consistent value. This pruning eventually triggers constraint propagations on other action variables domains.

The second plan is also interesting. It says: *"set the* runlevel *of machine* m *from the current one, i.e.* 3*, to another one* X *for which a file* TMAboot *exists in the location* /sbin/rcX*, then reboot the system"*. The proper values for X are obtained by means of a constraint associated to the precondition of action reboot.

Just to give an idea of how action preconditions look like, those associated to action reboot(*<machine>* m, *<at_runlevel>* I, *<TMAboot_path>* X) are:

runLevel(*<machine>* m,*<runlevel>* I),
inDirectory(*<file_name>* TMAboot,*<path>* X),
configDir(*<path>* X,*<runlevel>* I).

It is worth noting that the planner does not commit to a fixed runlevel: this is a good point, because any of them actually suits the application. This fact shows how the possibility of working with domains rather than with fully specified values allows the planner to produce more expressive plans, with respect to the problem they are solving. Saying *"start it at* another *runlevel"* is more appropriate and less restrictive than just saying *"start it at* exactly that *runlevel"*.

Now, in order to clarify the mechanism of labelling and data acquisition, let us suppose that the system operator has chosen the second plan. In the domain of variable X, we have two consistent values, i.e., /sbin/rc2 and /sbin/rc5. In addition, we leave the domain open in order to take into account non monotonic changes. Thus, the domain of variable X becomes Dom(X)=[/sbin/rc2, /sbin/rc5 | X']. Let us suppose that in the meanwhile another process moved file TMAboot from /sbin/rc2 to /sbin/rc1 and deleted the directory /sbin/rc5 with all its contents. The executor should check if the domains are still non-empty through IC verification, possibly prune them and eventually perform the labelling. If some values are still consistent with the precondition, the action can be executed. Unfortunately, in this case, the precondition is not satisfied neither for /sbin/rc2 nor for /sbin/rc5: a new acquisition, at execution time, is needed, in order to give X' one or more consistent values. Fortunately, it results in a non-empty set, containing X'=[/sbin/rc1]. Labelling produces the action copy(*<file_name>* TMAboot, *<from_path>* /sbin/rc1, *<to_path>* /sbin/rc3) and the executor performs it. It is worth noting that there is no need for re-planning, even in case the planner is not the only cause of change of the system: instead of asking the planner for another problem to solve, the executor needs only to check the consistency of the domain associated to variable X and to pick up another value from it.

[6]We refer to both components because they share the interactive constraints and associated domains.

7 Related Work and Conclusions

This paper discussed the problem of plan construction and execution in dynamic environments. A classification of the approaches to planning in presence of uncertainty due to dynamic environment is given in [10], where the authors distinguish among *conditional*, *assumption-based* and *deferring* planners. Our approach fits in the third planning strategy, which consists in delaying data acquisition as much as possible, interleaving it with action execution. In this way, it is possible to both minimize the amount of managed data and reduce the gap between the world as it is observed at planning time and the world the executor acts on. Our deferring planner is based on the Interactive Constraint Satisfaction framework [7], which represents an extension of the CS framework based on Interactive Constraints able to acquire consistent knowledge during their propagation.

Among the previous (*deferred planning*) approaches, XII [3] and Sage [4] are worth mentioning. XII (eXecution and Incomplete Information) is a planner thought for controlling an Internet softbot, which works with local closed world information. It manages incomplete information by means of least redundant sensing actions introduced at execution time. In particular, it is assumed that some information is possibly missing, but that the information that is known is correct. That approach is different from ours, because we do not make any assumption on the correctness of knowledge. In general, our idea is more oriented to dynamic environments, where knowledge might have a non monotonic growth.

Sage is a planner addressing the issue of interleaving sensing with execution in order to cope with incomplete knowledge. It is an integrated tool in which planning, sensing and replanning can be performed during execution. Its application domain is information retrieval over distributed networks. Such environment allows the system to tightly integrate planning and execution. In particular the execution of each individual action is a necessary step to complete the plan. In case of execution failures, it will re-build only portions of the plan in order to maintain as much of the execution plan as possible. This makes a difference between the two scopes, the one of Sage and ours. In particular, we do as much preplanning as possible. In case of failure either we refine the current plan, as in Sage, or we perform necessary backtracking steps and (possibly partial) replanning. Finally, we need backup operations for backtracking from *irreversible* actions. This kind of actions does not exist in Sage application domain.

As far as the other planning strategies are concerned, CNLP [9], represents an example of planner following the *planning for all contingencies* strategy. Conditional planners build a conditional tree of subplans at planning time, anticipating all the possible failure conditions and building suitable repair subplans in advance. They take advantage of observations made during plan execution to select the actions that have to be executed. Conditional plans generated by CNLP are provided with *conditional links* describing at each step which action to execute, depending on the outcome of the previous one.

Assumption-based planners like BURIDAN [5] use a probability distribution over possible worlds and over possible changes to the worlds, in order to build a subset of the whole conditional tree, and in particular the one corresponding to the most probable plan.

C-BURIDAN [2] adopts a hybrid approach, extending the BURIDAN probabilistic action representation to allow actions with both informational and causal effects; it builds plans on the CNLP algorithm.

Finally, in this paper we faced the problem of non monotonic changes. When domain values are no longer consistent during plan execution, open domains allow the executor to retrieve new consistent values and to execute actions on them, thus earning in efficiency during a backtracking step.

We are currently defining a classification of action preconditions and involved resources aimed at defining the best way of performing acquisition as far as the amount of retrieved data and the acquisition time are concerned. The classification is based on the following parameters: (*i*) the data production activity cost, i.e., how much does it cost for the distributed system to provide required values; (*ii*) the communication cost between the planner and the underlying distributed system and (*iii*) the frequency of change of the retrieved data, i.e., volatile vs. stable resources.

8 Acknowledgments

Authors' work has been partially supported by Hewlett Packard Laboratories of Bristol-UK (Internet Business Management Department) and CNR (Project 40%).

References

[1] R. Barruffi, E.Lamma, P.Mello, and M. Milano. Least commitment on variable binding in presence of incomplete knowledge. In *Proceedings of the European Conference on Planning (ECP99)*, 1999.

[2] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of AIPS-94*, 1994.

[3] K. Golden, O. Etzioni, and D. Weld. Xii: Planning with universal quantification and incomplete information. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning, KR'94*, 1994.

[4] C.A. Knoblock. Planning, executiong, sensing, and replanning for information gathering. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995.

[5] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of AAAI-94*, 1994.

[6] E. Lamma, M. Milano, R. Cucchiara, and P. Mello. An interactive constraint based system for selective attention in visual search. *Proceedings of the ISMIS'97*, 1997.

[7] E. Lamma, M. Milano, P. Mello, R. Cucchiara, M. Gavanelli, and M. Piccardi. Constraint propagation and value acquisition: why we should do it interactively. *Proceedings of the IJCAI*, 1999. to appear.

[8] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, 1990.

[9] M.Peot and D. Smith. Conditional nonlinear planning. In *Proceedings of 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, 1992.

[10] D. Olawsky and M. Gini. Deferred planning and sensor use. In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[11] D.S. Weld. An introduction to least commitment planning. *AI Magazine*, 15:27–61, 1994.