[Spalazzi *et al.*, 1992] L. Spalazzi, A. Cimatti, and P. Traverso. Implementing planning as tactical reasoning. In *Proc. AIS'92, AI Simulation and Planning in High Autonomy Systems Conference*, pages 80–85, Perth Australia, 1992. IEEE Computer Society Press. Revised version of Technical Report 9104-23, IRST, Trento, Italy.

[Stefik, 1981] M. J. Stefik. Planning and Meta-Planning. *Artif. Intell.*, 16:141–169, 1981.

[Stringa, 1990] L. Stringa. An integrated approach to artificial intelligence. Technical Report 9012-11, IRST, 1990.

[Swartout, 1988] W. Swartout. DARPA Santa Cruz Workshop on Planning - Workshop Report. *The AI Magazine*, 9(2):115–130, Summer 1988.

[Traverso *et al.*, 1992] P. Traverso, A. Cimatti, and L. Spalazzi. Beyond the single planning paradigm: introspective planning. In *Proceedings ECAI-92*, pages 643–647, Vienna, Austria, 1992. IRST-Technical Report 9204-05, IRST, Trento, Italy.

[Wilensky, 1979] R. Wilensky. Meta-Planning: Representing and Using Knowledge About Planning in Problem Solving and Natural Language Understanding. In A. Collins and E. Smith, editors, *Readings in Cognitive Science. A Perspective from Psycology and Artificial Intelligence*. Morgan Kaufmann, 1979.

[Wilkins, 1988] D. E. Wilkins. *Practical Planning: extending the classical AI planning paradigm*. Morgan Kaufmann, San Mateo, 1988.

[Giunchiglia *et al.*, 1991] F. Giunchiglia, P. Traverso, A. Cimatti, and L. Spalazzi. Programming planners with flexible architectures. Technical Report 9112-19, IRST, Trento, Italy, 1991. Revised version entitled: "Tactics: extending the notion plan" presented at the ECAI-92 workshop Beyond Sequential Planning (Vienna, August 1992).

[Giunchiglia, 1992] F. Giunchiglia. The `GETFOL` Manual - `GETFOL` version 1. Technical Report 9204-01, DIST - University of Genova, Genoa, Italy, 1992. Forthcoming IRST-Technical Report.

[Gordon *et al.*, 1979] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[Hammond, 1990] K. J. Hammond. Explaining and Repairing Plans that Fail. *Artif. Intell.*, 45(1-2):173–228, 1990.

[Hanks and Firby, 1990] S. Hanks and R. J. Firby. Issues and Architectures for Planning and Execution. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 59–70, 1990.

[Hayes-Roth, 1985] B. Hayes-Roth. A Blackboard Architecture for Control. *Artif. Intell.*, 26:251–321, 1985.

[Kaelbling, 1987] L.P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning about actions and plans: Procedings of the 1986 Workshop*. Morgan-Kaufmann Publishers, 1987.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. *Artif. Intell.*, 33(3):1–4 64, 1987.

[Russel and Wefald, 1989] S. Russel and E. Wefald. Principles of Metareasoning. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, 1989.

[Schoppers, 1987] M. J. Schoppers. Universal plans for Reactive Robots in Unpredictable Environments. In *Proc. of the 10th International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[Simmons, 1990] R. Simmons. An Architecture for Coordinating Planning, Sensing and Action. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 292–297, 1990.

[Simmons, 1991a] R. Simmons. Concurrent planning and execution for a walking robot. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2086–2091, Sacramento, CA, 1991.

[Simmons, 1991b] R. Simmons. Coordinating planning, perception and action for mobile robots. In *AAAI Spring Symposium on Integrated Intelligent Architectures*, pages 156–159, 1991. SIGART Bulletin vol. 2 no. 4.

sium on Artificial Intelligence and Mathematics, Fort Lauderdale - Florida, January 1992. To appear in: The Annals of Artificial Intelligence and Mathematics.

[Beetz and McDermott, 1992] M. Beetz and D. McDermott. Declarative Goals in Reactive Plans. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proc. of 1st Internationa l Conference*, pages 3–12, San Mateo, CA, 1992. Morgan Kaufmann.

[Brooks, 1986] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[Cimatti *et al.*, 1992] A. Cimatti, P. Traverso, S. Dalbosco, and A. Armando. Navigation by Combining Reactivity and Planning. In *Proc. Intelligent Vehicles '92*, Detroit, 1992. IRST-Technical Report 9205-11, IRST, Trento, Italy.

[Durfee and Lesser, 1986] E.H. Durfee and V.R. Lesser. Incremental Planning to Control a Blackboard-based Problem Solver. In *Proc. of the 5th National Conference on Artificial Intelligence*, Philadelphia, PA, 1986.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of Theorem Proving to Probl em Solving. *Artif. Intell.*, 2(3-4):189–208, 1971.

[Firby, 1987] R. J. Firby. An Investigation into Reactive Planning in Complex Domains. In *Proc. of the 6th National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA, USA, 1987.

[Firby, 1992] R. J. Firby. Building Symbolic Primitives with Continuous Control Routines. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proc. of 1st International Conference*, pages 62–69, San Mateo, CA, 1992. Morgan Kaufmann.

[Georgeff and Lansky, 1987a] M. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proc. of the 6th National Conference on Artificial Intelligence*, pages 677–682, Seattle, WA, USA, 1987.

[Georgeff and Lansky, 1987b] M. P. Georgeff and A. L. Lansky. Procedural Knowledge. Technical Report 411, A.I. Center, SRI International, Menlo Park, California, 1987.

[Georgeff, 1989] M. Georgeff. An embedded reasoning and planning system. In J. Tenenberg, J. Weber, and J. Allen, editors, *Proc. from the Rochester Planning Workshop: from Formal Systems to Practical Systems*, pages 105–128, Rochester, 1989.

[Georgeff, 1991] M. P. Georgeff. Situated Reasoning and Rational Behaviour. Technical Report 21, Australian AI Institute, Carlton, Victoria, Australia, 1991.

[Giunchiglia and Traverso, 1991] F. Giunchiglia and P. Traverso. Reflective reasoning with and between a declarative metatheory and the implementation code. In *Proc. of the 12th International Joint Conference on Artificial Intelligence*, pages 111–117, Sydney, 1991. Also IRST-Technical Report 9012-03, IRST, Trento, Italy.

[Giunchiglia and Traverso, 1992] F. Giunchiglia and P. Traverso. A Metatheory of a Mechanized Object Theory. Technical Report 9211-24, IRST, Trento, Italy, 1992.

[Giunchiglia and Traverso, 1993] F. Giunchiglia and P. Traverso. Program tactics and logic tactics. Technical Report 9301-01, IRST, Trento, Italy, 1993.

implemented naturally in MRG.

Notice that the similarity of MRG with the conventional programming languages (*e.g.* Pure LISP) is only apparent and syntactical. The building blocks of the MRG language are primitive tactics. They represent basic planning activities. We have circumscribed the constructs of a programming language that are needed for building planners. For instance `orelse` allows us to handle failure. Furthermore, MRG provides an architecture to manage tactics, to relate tactics to goals and facts and to activate tactics to respond to goals and facts. These mechanisms are not provided by general purpose programming languages. Finally, and perhaps most importantly for the practical use of MRG, the level of abstraction is much higher than in a general purpose programming language. Within MRG, the user has not to develop the application from scratch.

7 Conclusions and future work

In this paper we have presented MRG, a domain independent system for the development of planners. We see MRG as a step towards general purpose systems for building real world applications, that is systems that allow the user to fully customize the whole planner depending on the application requirements. MRG is at the moment used in a complex real world application under development at IRST.
Future developments of this research include a general purpose definition of the interactions between parallel tactics and a formalization of the MRG language semantic. A question may be how to define theoretically and generally a way to generate and reason about tactics. Whereas it has been shown how to reason about classical plans, it is not obvious how to reason about tactics. In [Giunchiglia and Traverso, 1991; Giunchiglia and Traverso, 1993; Giunchiglia and Traverso, 1992] it has been shown how a first order formalized metatheory can be used to represent expressions that have some similarities with MRG tactics. This metatheory is able to represent failure. The idea is to translate tactics and tacticals into terms of the logical metatheory. Thus a logical metatheory can be used to reason about, prove properties of and automatically generate tactics via metalevel theorem proving. We plan to start a research project that aims at giving a formal definition of the mapping between MRG tactics and terms of the metatheory described in [Giunchiglia and Traverso, 1991; Giunchiglia and Traverso, 1993]. One of the reasons to represent failure in a logical metatheory is the ability to prove *a priori* when a tactic is likely to fail. In section 3.3, we have explained how the tactical `orelse` is useful to deal with failures. But critical and unrecoverable failures should be avoided. Reasoning by deduction in a logical metatheory allows the planner to state that, according to the model of the world axiomatized in the metatheory, the plan fails. This allows the planner to be very careful in executing dangerous tactics.

References

[Agree and Chapman, 1987] P. Agree and D. Chapman. Pengi: an implementation of a Theory of Activity. In *Proc. of the 6th National Conference on Artificial Intelligence*, pages 268–272, Seattle, WA, USA, 1987.

[Armando and Giunchiglia, 1992] A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. In *Second International Sympo-*

7 we briefly hint how we are planning to deal with the problem of reasoning about tactics.

The systems closest to MRG are the *situated planners* [Agree and Chapman, 1987; Beetz and McDermott, 1992; Firby, 1992; Georgeff, 1991; Hayes-Roth, 1985; Laird *et al.*, 1987; Simmons, 1991a]. All these systems address the issue of integrating reactivity and reasoning capabilities, even if each of them uses quite different techniques.
RAP [Firby, 1987; Firby, 1992; Hanks and Firby, 1990] is used in a "partitioned architecture" where a strategic planner interacts with a completely reactive system (RAP Executor) through a shared world model and plan representation. One of the advantages of this architecture is the fact that computational costs are reduced with respect to "uniform architectures". Tactics implement a uniform architecture since they can contain (a combination of) deliberation and action. However RAP does not address the problem of having a flexible architecture where control strategies can be defined. Several control mechanisms of RAP are hardcoded. The whole execution system implements a particular strategy that might not be suited to all applications. For instance, dealing with failures is performed by trying different strategies (called methods) that are applicable till the same strategy is applied twice. In MRG no failure handling mechanism is hardcoded. A tactic can specify to try different strategies, to try the same strategy n times, to replan etc.

XFRM [Beetz and McDermott, 1992], depending on the time constraints, executes either default plans or new plans obtained by transforming the default plans by means of heuristic rules. The plan transformation rules are features external to the plan. The idea of MRG is to provide a language where plan manipulations are explicitly represented within tactics. However, the plan manipulation tactics that we have implemented are still simple ones. We plan to test this capability with further application development.

The motivations underlying TCA [Simmons, 1990; Simmons, 1991a; Simmons, 1991b] are very close to the motivations underlying MRG. TCA plans (called Task Trees) provide a language for representing the various planning activities: plan formation, plan execution, information acquisition and monitoring. The system can flexibly perform interleaving planning and execution, run-time changing of a plan and coordinating multiple tasks. Similarly to MRG, the TCA exception handling facilities support context-dependent error recovery since different error handlers can be associated to different nodes in the task tree. Nevertheless, error handlers are not written explicitly in the TCA planning language. In MRG, failure capturing is explicitly represented by the construct `orelse`. Recovery from failure is uniformly represented by tactics. This allows the user to define plans and failure handling mechanisms with the same language and to flexibly modify them.

In PRS [Georgeff, 1989; Georgeff and Lansky, 1987a; Georgeff, 1991; Georgeff and Lansky, 1987b], plans (called KAs) describe how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations. Metalevel KAs encode various methods for chosing among multiple applicable KAs. They provide a high amount of flexibility in forming plans. The same amount of flexibility is provided in MRG by tactics. In PRS, plan execution and plan formation are not explicitly denoted by different tactics, like `plan-for` and `exec` (see section 4). This distinction is explicit in MRG tactics; it opens up the possibility to reason about how plan formation and plan execution can be performed to solve a problem; it opens up the possibility to reason about when it is better to generate a plan or to execute a plan. As a further consequence of this fact, different planning techniques can be

```
(deftac embedded-replan (location old-plan)
  (then
   (set-node-inaccessible (query-position) old-plan)
   (embedded-navigation-to location)))
```

where, `perceive-fixed-obstacle` acquires information from the real world to decide whether to replan or to execute the plan again. `wait` is a primitive tactic that stops the robot for a while.
`classic-navigation-to` and `embedded-navigation-to` are two examples of tactics that represent different navigation control mechanisms. The full set of tactics (whose description is out of the goals of this paper) constitutes a navigation supervisor in `MRG` that performs different behaviours in different situations. For instance, depending on the area of the building where the robot navigates, that may be usually more or less crowded, the system activates either `classic-navigation-to` or `embedded-navigation-to`. The system plans paths in different modalities, for instance by using `shortest-nav-plan-for` where the navigation system is supposed to be robust and using `easiest-nav-plan-for` where navigation is difficult. In critical situations it performs executions with a high degree of monitoring to detect navigation problems. Under user request, the system can be highly interactive, for instance splitting plans in subplans and asking for new information at each step. We have still to define strategies where the plan formation mechanism is modified at real time according to the facts that get known to `MRG`. But we should have the right building blocks to deal with this problem.

6 Related work

In most of the existing planning systems the planning activities and the control mechanism are fixed in the implementation code. They may be well suited to solve some classes of problems, but it may be difficult to use them in complex real world large-scale applications, where different requirements have to be fulfilled by one system. For instance, classical planners, case-based, conditional, deferred planning techniques have problems to deal with domains that are both dynamic and unpredictable, while reactive systems seem not to be able to perform high level behaviours where planning ahead is required. `MRG` tactics, as shown in section 4, allow us to mix different planning techniques according to the application.

The idea of reasoning about the plan formation step is known as *metaplanning*, see for instance [Hayes-Roth, 1985; Laird *et al.*, 1987; Stefik, 1981; Wilensky, 1979]. Metaplanning techniques do not provide a metalevel programming language to reason about all the planning steps (execution, monitoring, failure handling, information acquisition etc.). All the basic planning activities of `MRG` are represented within `MRG` itself. This opens up the possibility to reason about all the different planning steps and not only about plan formation. This is very important in dynamic and unpredictable environments where all the problems cannot be solved at planning time.

Decision analytic techniques (see for instance [Russel and Wefald, 1989]) allow for reasoning about alternative actions, deliberation and execution. The main problem of this approach is to build the right utility functions for any application. Our approach is different. We give the user a general purpose language to define and to generate its own strategies. In section

5.3 Some different control mechanisms in MAIA

Given these libraries, we have implemented tactics that allow the robot to perform different navigation behaviours in different situations. We give some examples below. In the first example we solve the navigation problem with a "classical-like" approach: we plan, execute and handle failure by replanning. We define the tactic `classic-navigation-to` with a tactic similar to the one described at page 11.

```
(deftac classic-navigation-to (location)
 (let (plan (nav-plan-for (robot-at location)))
  (orelse
   (exec-nav-plan plan)
   (if (node-in-path? (query-position) plan)
       (then (set-node-inaccessible (query-position) plan)
             (classic-navigation-to location))
       (classic-navigation-to location)))))
```

The tactic above plans for a path and executes the plan. The tactic deals with two different forms of failure[6]. If execution fails, the tactic tests whether the robot has been stopped along the planned path (`node-in-path?`). If it is the case, the cause of failure might be a not avoidable obstacle along the way. In this case the tactic sets the node in the map where the robot has stopped as not accessible and replans. If the robot has moved out of the planned path, since, for instance, the robot has missed a landmark, then the robot plans simply for a new path from the position where it has moved to.

In a dynamic environment like a lab, where obstacles are usually people moving and standing in corridors, a tactic like this forces the robot to find alternative paths continuously even when moving obstacles are found along the way. We thus define a tactic that interleaves plan formation, execution and real world acquisition. This tactic, when a moving obstacle is found on the way, does not necessary replan for a new path, but it waits for a while and tries again. This sounds reasonable when we have mobile obstacles in the building. In this case a particular strategy is executed to react to failure before replanning:

```
(deftac embedded-navigation-to (location)
 (let (plan (nav-plan-for (robot-at location)))
   (orelse
     (exec-nav-plan plan)
     (if (perceive-fixed-obstacle)
         (embedded-replan location plan)
         (then
            (wait)
            (orelse
              (exec-nav-plan (path-to-end plan (query-position)))
              (embedded-replan location plan)))))))
```

[6]This tactic is a simplified version of the real tactic that handles failure by replanning in the application. The tactic in the application distinguishes many different forms of failures and handles them in different ways. The goal of this and the next example is to give an idea of how different control mechanisms are used in this application.

- `common-subpath`, which returns the initial subpath common to the paths given in input;
- `rectify-path`, which splits a path into a sequence of paths which do not contain turns;
- `path-to-end` which, given a path and a node, returns the path from the node to the end.

The MAIA plan execution library

Navigation paths can be executed according to the following execution tactics.

- `exec` is the general purpose plan execution mechanism that calls the tactic interpreter.
- `exec-nav-plan` is an interpreter dedicated to navigation plans. It translates the navigation plan in a sequence of landmarks, sends the sequence to the modules controlling the sensors and actuators of the mobile robot and receives informations about the navigation execution.
- `exec-monitor-node` is a tactic that, given a tactic representing a path as input, executes the tactic by monitoring information each time the robot navigates through a new node in the map.
- `exec-monitor-distance` is a tactic that executes a tactic and monitors the position of the robot every time the robot has navigated for a given distance.
- `exec-monitor-time` is a tactic that executes a tactic and monitors the position of the robot at each given time interval.
- `exec-monitor-pos` is a tactic that executes a tactic and monitors whether the robot has reached a given position.

Furthermore, in this library there are different execution modalities (some of them mentioned in section 3.2), like `simulate-nav-plan`, `display-exec-nav-plan` and `display-simulate-nav-plan`.

The MAIA acquisition library

The acquisition library provides tactics that get information about the real world.

- `perceive-fixed-obstacle`, which activates a module that tries to detect the presence of fixed obstacles on the way;
- `perceive-crowding`, which activates a module that tries to detect the presence of crowding;
- `query-position`, which returns the node in the map representing the current position of the mobile robot;
- `query-state`, returning the current state of navigation (e.g. navigating, suspended, success, failure);
- `query-failure-description`, returning a description of the failure provided by the modules controlling robot sensors and actuators.

This library implements a communication protocol between `MRG` and the navigation system on board of the robot.

sequences of navigation steps, like

```
(then
 (move-to room-100)
 (then (move-to room-200)
       (move-to room-300)))
```

where `move-to` is a tactic that when executed moves the robot from the robot current position to a position in the adjacent node specified by the argument. We refer to these plans as navigation paths. Navigation paths can be interleaved with other kinds of tactics, like tactics that acquire information from the real world. For instance

```
(then
 (move-to room-100)
 (if (perceive-fixed-obstacle)
     (move-to room-300)
     (move-to room-400)))
```

where `perceive-fixed-obstacle` is an acquisition tactic that returns true when the robot recognizes a fixed obstacle in front of itself. If the robot cannot activate its sensors, then the tactic fails.
Some of the navigation plan formation tactics are listed below.

- `shortest-nav-plan-for` returns the geometrically shortest path;
- `least-crowded-nav-plan-for` returns the best path according to the crowding cost;
- `easiest-nav-plan-for` returns the best path according to the navigation cost;
- `least-nav-plan-for` returns the best path according to the cost function provided as input, which can be a combination of all the costs contained in the map;
- `nav-plan-for` returns a path according to a combination of all the different costs.

More complex tactics search for all possible paths between two locations, and determine paths satisfying constraints of different forms. Some of them are:

- `all-paths`, which returns a list of the possible paths from starting to target position;
- `ord-constrained-paths`, returns the list of all the paths from starting to target position, constrained to pass through a list of specified locations, according to a supplied ordering relation;
- `unord-constrained-paths` returns the list of all the paths from starting to target position, constrained to pass through a list of specified locations, in the most convenient order.
- `shortest-ord-constrained-path` returns the geometrically shortest path between the ones satisfying the set of ordered constraints specified.

Furthermore, we have plan manipulation tactics that take a navigation plan as argument and return a tactic. Some of them are listed below.

A system with a fixed control structure would be inadequate to deal with such an application domain: indeed it has very different requirements depending on the tasks to be achieved. The system must both reason to solve high level tasks and perform reactive behaviours to deal with an unpredictable and dynamic environment.
In this section we focus on the part of `MRG` [Cimatti *et al.*, 1992] that supervises the navigation system of MAIA, a reactive system controlling a mobile robot provided with special purpose vision and ultrasound sensing systems.

5.1 The MAIA knowledge bases

The MAIA knowledge base `map-kb` contains a map of the building. The map contains information describing various aspects of the navigation domain. The map is structured as a graph, whose nodes represent areas of the building and whose arcs represent the connections between them. Nodes can contain object descriptions, like desks and printers, and landmarks. Each node, object and arc contains geometrical information: its coordinates, its dimensions and so on. Nodes are defined and classified according to their features as "corridor", "open space", "room" and "cross"; similarly, arcs are classified as "open" and "door". Each arc and node have associated a set of different costs. Some examples are "geometrical cost" (function of the distance between the nodes), "navigation cost" (for instance, curves are more "expensive" than aligned nodes, rooms with a lot of obstacles are more expensive than empty rooms), and "crowding cost" (crowded areas are more difficult to navigate). The map is also used to store other information, which is acquired during navigation: *i.e.*, descriptions of a navigation sessions, success or failure, obstacle descriptions, and so on.

The information contained in `map-kb` can be accessed by means of the suitable knowledge base management tactics (*i.e.* `arc-get-type`, `node-get-objects`, `node-get-crowding`). It is possible, for instance, to enquire the knowledge base to know whether a node is along a navigation path in the map (`node-in-path?`). Primitive tactics associate informations acquired during a navigation session with nodes and arcs (*e.g.* `node-add-failure-description`). Other information stating if a node is believed to be accessible or not can be set according to the current state of navigation (`set-node-inaccessible`) and accessed during the planning process (`node-get-accessible-nodes`).

The MAIA knowledge base `robot-kb` contains information about the robot (the robot features, the robot type, its communication mode etc.), its navigation status, (like navigating, suspended, success, failure, the current robot position), the current goal and the current navigation plan. These information are used during navigation and failure handling. Primitive tactics like `get-current-goal`, `get-current-plan`, `get-current-position` and `get-current-failure` access the `robot-kb`; primitive tactics like `set-current-goal`, `set-current-plan`, `set-current-position` and `set-current-failure` update the `robot-kb`. The knowledge base is updated with the information acquired from the real world by tactics in the acquisition libraries (see page 17).

5.2 The MAIA libraries

The MAIA plan formation library

The plan formation library contains tactics that generate navigation plans. Navigation plans are tactics that specify a navigation that the robot has to perform. They are in general

plans for moving the robot to the door of the office where the desk is located [`(map-door-of (map-room-of desk-loc))`].
The planning to reach the desk location `(nav-plan-for (robot-at desk-loc))` is then deferred after the planner knows whether the door of the office is opened. `(door-closed?)` is an acquisition tactic that activates a module controlling a sonar sensor. If the door is closed, `open-door` is a primitive tactic whose task is to open the door.
Reactive planning systems provide the ability to react immediately to environmental changes. In reactive systems, plan formation can be seen as the interpretation of an external stimulus plus the selection of an adequate precompiled reaction. Plan execution is the reaction itself. Failures are treated like any other stimulus, thus replanning is not needed. In `MRG` a reactive system can be represented by the following tactic[5]:

```
(deftac reactive-loop ()
 (then (exec (select-reaction (get-stimulus)))
   (reactive-loop)))
```

where `reactive-loop` is a recursive tactic implementing a reactive loop, `get-stimulus` is a primitive tactic linked to a sensor controller and `select-reaction` selects an adequate reaction to the stimulus.
So far we have shown how `MRG` can be used to implement different planning techniques. Moreover, different planning techniques can be alternatively used and combined according to the particular application requirements. As a simple example of this ability, we show how to implement a control mechanism that intermixes a reactive and a classical planning technique. For instance, we can plan ahead whenever no immediate reaction to an external event can be performed by our reactive system (*e.g.* in the case the external event is the request of a high level task that needs some search):

```
(deftac mixed-planning ()
 (let (input (get-stimulus))
  (then (exec (orelse (select-reaction input)
                      (plan-for input)))
        (mixed-planning))))
```

In the tactic above, if no precompiled reaction to the stimulus memorized in the variable `input` is available, then `(select-reaction input)` fails; `orelse` captures failure and reasons about the goal to be achieved by generating a plan (`(plan-for input)`) and executing the plan.

5 An application in MRG

At the moment we are experimenting with `MRG` in a complex large-scale application (called MAIA) under development at IRST [Stringa, 1990]. The objective of the MAIA project is to build an intelligent robot able, among other things, to navigate in an unpredictable environment, to interact and exchange information with people, to find persons in the IRST building and to help them. Different capabilities, like navigation, vision capabilities, voice recognition and synthesis, natural language understanding are needed to build MAIA.

[5]Notice that goals and stimuli can be given in input to `MRG` asynchronously. Different reactive loops can be active at the same time.

For instance, most *classical planners* perform a phase of plan generation followed by plan execution. In case of failure during plan execution, they replan again until the execution succeeds or no plan can be generated anymore. In MRG a simple example of a classical planner can be implemented by the following tactic[3]:

```
(deftac classic-planner (goal)
 (let (plan (plan-for goal))
  (orelse (exec plan)
   (then (modify-planner-status)
         (classic-planner goal)))))
```

where plan-for and exec are a plan formation and a plan execution tactic respectively. modify-planner-status is a knowledge base management tactic that updates the knowledge base containing an internal representation of the world.
In *case-based planners*, plan formation is performed by using and debugging known plans. Once a planner has constructed a useful plan, the result is stored for later use. An example of case-based planning is shown by the following tactic[4]:

```
(deftac case-based-planner (goal)
 (exec (orelse (retrieve-tac goal)
               (learn-tac (plan-for goal)))))
```

where retrieve-tac is a plan generation tactic that searches for and debugs a tactic in a library and learn-tac is a tactic that updates the library and returns its argument (the plan generated by plan-for).
The plan formation phase of *conditional planning* involves "generating alternative plans for each possible outcome or world state, including an appropriate test in the plan to choose from the alternatives" [Swartout, 1988]. Conditional plans can be obviously expressed within the MRG language by means of the tactical if.
Deferred planning systems, during plan formation activity, can defer plan generation, until some information is available, by interleaving plan formation and plan execution. Within MRG it is possible to interleave plan formation and execution to acquire information at execution time as shown by the following example.

```
(deftac deferred-navigation-to (desk-loc)
 (then
  (exec-nav-plan
   (nav-plan-for (robot-at (map-door-of (map-room-of desk-loc)))))
  (if (door-closed?)
      (then (open-door)
            (exec-nav-plan (nav-plan-for (robot-at desk-loc))))
      (exec-nav-plan (nav-plan-for (robot-at desk-loc))))))
```

The tactic deferred-navigation-to is a strategy to let the robot reach a desk location [desk-loc] in an office. Rather than planning for going to the desk location, the tactic

[3] At page 11 we have shown a tactic that is an example of a classical planner for the robot navigation.

[4] For simplicity, in this and in the next example, we consider neither replanning nor calling recursively the planner itself.

In the example above, when `orelse` "captures" failure, the reaction to failure is not replanning, but the execution of an exception handling routine. `EH-off` is a constant denoting a tactic that represents an emergency plan. `exec-EH` is an execution tactic.
We can think of cases where we need replanning and cases where we need precompiled plan execution.

```
(then
 (let (safe-plan (nav-plan-for (robot-at loc1)))
   (orelse (exec-nav-plan safe-plan)
           (then (set-node-inaccessible (query-position) safe-plan)
                 (exec-nav-plan (nav-plan-for (robot-at loc1))))))
 (let (danger-plan (nav-plan-for (robot-at loc2)))
   (orelse (exec-nav-plan danger-plan)
           (exec-EH EH-off))))
```

The first plan moves the robot to the location represented by the variable `loc1`. The plan is stored in the variable `safe-plan`. It is considered safe and then failure is handled by replanning. The second plan moves the robot to the location represented by the variable `loc2`. The plan is stored in the variable `danger-plan`. Failure while executing this plan is considered to be a dangerous situation the robot should not reason about. The robot executes a precompiled emergency routine that turns it off.
In conclusion, `MRG` does not provide any hardcoded failure handling mechanism. Its language is provided with the basic control structure to define failure handling according to the situation and the application requirements.

`MRG` features a definition mechanism to define new tactic identifiers through the construct `deftac`. We write as $\tau[x_1, ..., x_n]$ a tactic τ where $x_1, ..., x_n$ are the variables appearing in τ such that they are not bound to a `let` tactical. We can define a tactic identifier t_i with the following expression.

$$\begin{array}{l}\texttt{(deftac}\ \ t_i\ \texttt{(}\ x_1\ ...\ x_n\ \texttt{)} \\ \qquad \tau[x_1\ ...\ x_n]\ \texttt{)}\end{array}$$

`deftac` supports definitions of recursive tactics.
Defined tactics can be added to the `MRG` libraries. The user is thus provided with the ability to incrementally build its own libraries. Defined tactics can extend the libraries with both basic planning activities and control mechanisms that combine basic planning steps. For instance, in the former case, a defined tactic can combine two or more primitive tactics that implement plan formation activities. In the latter case, it is possible to combine execution tactics with acquisition tactics to obtain monitored execution; it is possible to combine plan formation, plan execution and information acquisition tactics to obtain tactics that realize deferred planning techniques. This issue is discussed more in depth in section 4.

4 Programming planning techniques

All the planning paradigms and techniques differ depending on their basic planning activities and control mechanisms [Traverso *et al.*, 1992]. That is, they differ depending on how the basic planning tasks are implemented and on how they activate, combine and control the various basic planning activities.

```
(let (start-pos (query-position))
 (then
  (exec-nav-plan (nav-plan-for (robot-at room-100)))
  (exec-nav-plan (nav-plan-for (robot-at start-pos)))))
```

The tactic above simply moves the robot to `room-100`, then moves the robot back to the point where the navigation started. The original position is obtained by the evaluation of the acquisition tactic `(query-position)`. It is stored in the variable `start-pos` of the `MRG` language. In this case the tactical `let` is needed since the robot position changes while the robot is moving.

Conditional plans are represented by means of the tactical `if`. For instance, depending on some particular conditions, it is possible either to execute a precompiled plan or to generate a new plan. Let us consider the following example:

```
(if (map-kb-modified?)
    (exec-nav-plan (nav-plan-for (robot-at area-east)))
    (exec-nav-plan nav-plan-east))
```

In the tactic above, the knowledge base management tactic `(map-kb-modified?)` is associated to an action testing whether the status of the map has been updated during a navigation session, for instance by adding new information about places where the robot should not have access. If the map has been modified, then it plans for a path to the location `area-east`, otherwise it moves the robot along the default path `nav-plan-east`.

The tactical `orelse` is the fundamental control structure for failure detection and for specifying reaction to failure. When `(orelse` τ_1 τ_2 `)` is interpreted, if τ_1 fails, the overall tactic does not necessarily fails: τ_2 is executed, and the overall tactic either fails or succeeds according to τ_2. Since τ_1 and τ_2 may implement any basic planning step, `orelse` can be used to specify different failure handling mechanisms. Let us consider the following example.

```
(let (plan (nav-plan-for (robot-at location)))
  (orelse (exec-nav-plan plan)
          (then (set-node-inaccessible (query-position) plan)
                (exec-nav-plan
                    (nav-plan-for (robot-at location))))))
```

The tactic above specifies that, if the robot navigation fails, then a new navigation plan is generated and executed. In this case we suppose that failure is due to an unpredictable obstacle on the way. `location` is a variable that gets instantiated to the place the robot has to reach; `set-node-inaccessible` is a knowledge base management tactic that adds new information to the map to take into account the new obstacle. In the tactic above, through the tactical `orelse`, we specify failure handling as replanning. But in certain situations there may not be enough time to generate a new plan. Think for instance of failures due to catastrophic situations in which the robot has quickly to turn off itself. In that case, an exception handling routine, *i.e.* a precompiled plan, should be promptly executed.

```
(let (plan (nav-plan-for (robot-at location)))
  (orelse (exec-nav-plan plan)
          (exec-EH EH-off)))
```

- `orelse`:

  Definition 3.3 (Orelse interpretation) *Let τ_1 and τ_2 be tactics. Let τ be the tactic* (`orelse` τ_1 τ_2)*. If the interpretation of τ_1 succeeds (with value v_1), then the interpretation of τ succeeds (with value v_1). If the interpretation of τ_1 fails, then the interpretation of τ is the interpretation of τ_2.*

  Notice that if τ_1 fails, then τ does not necessarily fails. In section 3.4, we show that `orelse` is the fundamental construct to deal with failure.

3.4 Extending the libraries

In this section, through some examples, we show how tacticals can be used to combine basic planning activities and to extend `MRG` libraries.

`then` is the tactical for the construction of sequential tactics. A simple example of sequential tactic is:

```
(then (follow-wall) (turn-right))
```

where `(follow-wall)` is a primitive tactic activating a navigation controlling system moving a robot along a wall in a building and `turn-right` is a tactic identifier associated to the robot operation of turning to a corridor on the right. The above tactic represents the sequence of robot actions: follow a wall (of a corridor) till the end, then turn right. This is a simple example of the usual notion of sequential plan, that is a sequence of actions in the external world. But sequential `MRG` plans can be tactics that represent sequences of basic planning activities, like plan formations and plan executions. For instance, the tactic below represents the sequence of the executions of two predefined (or previously generated) plans:

```
(then (exec-nav-plan charge-hall-nav-plan)
      (exec-nav-plan hall-charge-nav-plan))
```

where `charge-hall-nav-plan` and `hall-charge-nav-plan` are two constants denoting two `MRG` tactics representing navigation plans that, when executed, move the robot from a particular area to another (for instance from the hall to the charging area and vice versa). We can think of them as sequences of actions in the external world of the kind described in the previous example.

Tactics can be composed (see the last inductive step in the definition 3.2). For instance, without considering replanning, we can represent a composition of planning activities resembling a simplified version of classical planners control mechanism: "plan for a given goal, then execute the plan":

```
(exec-nav-plan (nav-plan-for (robot-at room-100)))
```

`MRG` plans are able to deal with dynamic environments. The execution of a plan may substantially differ in different situations. The `MRG` tactical `let` provides the ability to capture information about the state of the world (the result of the evaluation of τ_1) in a particular moment (situation) and eventually use it in the rest of the plan. For instance, `let` can be used to store information processed by the sensor controllers of a robot and that are likely to change (like the position of a moving object over time).

[Armando and Giunchiglia, 1992]. They constitute a set of general purpose deduction modules to add facts to the knowledge base. Domain specific knowledge bases can be accessed and updated by tactics linked to the suitable modules. For instance, primitive tactic identifiers can be linked to modules implementing the algorithms that access the nodes and the arcs of a topological graph representing the map of a particular navigation domain.
Notice that knowledge base management tactics implement the operations usually performed over knowledge bases. They are intrinsically different from other tactics since they are unlikely to fail.

Acquisition library. These tactics acquire information from the real world. For instance, they may activate systems controlling sensors in a robot or man-machine interface functions. Thus `MRG` tactics, beyond the capability to exploit the model of the world with knowledge base management tactics, have the capability to exploit information provided by the real world itself. For instance, `query-position` is a tactic acquiring information about the actual position of a robot. `query-failure-description` returns a description of the failure that occurred while a robot was navigating.

3.3 Compound tactics

In `MRG`, the various planning activities can be flexibly controlled by means of more complex tactics which can be obtained by combining primitive tactics in different manners.

Definition 3.2 (Tactic)

- *A primitive tactic is a tactic.*
- *Let* τ_1 *and* τ_2 *be tactics. Then* `(then` τ_1 τ_2`)` *is a tactic.*
- *Let* x *be a variable. Let* τ_1 *and* τ_2 *be tactics. Then* `(let (`x τ_1`)` τ_2`)` *is a tactic.*
- *Let* τ_1, τ_2 *and* τ_3 *be tactics. Then* `(if` τ_1 τ_2 τ_3`)` *is a tactic.*
- *Let* τ_1 *and* τ_2 *be tactics.* `(orelse` τ_1 τ_2`)` *is a tactic.*
- *Let* $(t_i\ \xi_1 ... \xi_n)$ *be a primitive tactic. The expression obtained by replacing any of* ξ_i *with a tactic is a tactic.*

`then`, `let`, `if` and `orelse` are symbols of the `MRG` alphabet. They are called **tacticals** and map tactics onto tactics. Sometimes we call tactics other than primitive tactics, *compound tactics*.
In the following we give a brief technical description of how tacticals are interpreted by the tactic interpreter.

- `then`: the argument tactics are executed in the given sequential order. If τ_1 fails, then τ_2 is not executed and the overall tactic fails. If τ_1 succeeds, then τ_2 is executed and the overall tactic succeeds or fails according to τ_2.
- `let`: the variable x is bound to the result of the evaluation of τ_1 and can be used within the tactic τ_2. A let tactic fails if and only if one of τ_1 and τ_2 fails, otherwise it succeeds.
- `if`: τ_1 is a tactic whose evaluation returns, when succeeds, a truth value. When a conditional tactic is executed, τ_1 is executed first; if it succeeds, τ_2 or τ_3 are executed according to the result. A conditional tactic fails if one of the executed tactics fails.

give an idea of how the user can use them as the set of building blocks to build a planner according to the particular application requirements.

Plan formation library. Plan formation (also called plan generation[2]) is the generation of a plan to achieve a given objective. In `MRG`, plan formation activities are represented by tactics that have a goal as argument and that return a tactic. These tactics can access the model of the world by means of the operations defined over knowledge bases. We can define tactic-action pairs that link tactic identifiers to general purpose search algorithms, like a state space search and a partial plan search. Tactics in this library can form a plan following a particular search strategy (*e.g.* depth first and breadth first searching strategies). The library can contain domain specific plan formation mechanisms, like tactics that generate a navigation plan for a robot by searching through a topological graph representing the map of the navigation domain. For instance, `nav-plan-for` is a plan formation tactic that, given a goal like `(robot-at room-100)`, returns a tactic that specifies a navigation plan (these kinds of tactics are described in more detail in section 5).
The plan formation library contains also plan manipulation tactics, that are tactics that perform manipulations over tactics. For instance, given a tactic that represents a navigation plan, a plan manipulation tactic can split the navigation in steps according to given criteria (examples of plan manipulation tactics are given in section 5, page 16).

Plan execution library. This library contains primitive tactics that take tactics as argument and execute them. In case of success, they return the result of the execution, otherwise they fail. The simplest execution tactic we can think of is a tactic that does nothing else but calling the tactic interpreter on its argument. In `MRG`, this tactic is called `exec`. The action linked to the tactic identifier `exec` is the `MRG` tactic interpreter itself. `exec` is useful in executing tactics stored in variables or denoted by constants. For instance, if a constant (or a variable) `t` denotes a tactic τ, `(exec t)` executes the tactic τ. `exec` is also useful in executing tactics that are results of plan formation activities. For instance, `exec` can execute a tactic returned by `nav-plan-for`. A tactic can always be executed by `exec`, since `exec` represents the tactic interpreter. But we can think of different ways of executing a plan. For instance, navigation plans can be executed by an interpreter dedicated to navigation. `exec-nav-plan` is a tactic identifier linked to an interpreter specific of navigation plans. Given a tactic generated by `nav-plan-for`, `exec-nav-plan` exchanges messages with the robot to execute the plan efficiently and effectively. If the suitable execution tactics are provided, the same plan can be executed by robots that have very different features. Several alternative execution modalities can be contained in this library. For instance, `simulate-nav-plan` is a tactic that interprets the navigation plan by sending data to a module simulating the robot; `display-exec-nav-plan` and `display-simulate-nav-plan` are execution tactics that display the results of the execution on a graphic map on the screen.

Knowledge Base Management library. The library contains tactics that access the knowledge bases. For instance, a first order logic knowledge base is accessed by tactics whose corresponding actions are the operations performed by a theorem prover. In this case, tactics access and add facts to the knowledge base. We are currently building a knowledge base derived from an interactive theorem prover, called `GETFOL` [Giunchiglia, 1992]. Tactics in this knowledge base are linked to `GETFOL` decision procedures for subsets of first order formulas

[2]Plan formation is sometimes simply called "planning". By "planning activities" we mean all the operations that a planning system performs and not only plan generation, *e.g* plan execution, monitoring, reaction to events etc.

are the basic building blocks of MRG plans. They are defined on the basis of goals, facts and symbols of the MRG language. Symbols are *tactic identifiers*, *constants* and *variables*. Tactic identifiers have associated an executable action type. For instance, `goto`, `put-block-on`, `nav-plan-for` and `exec-nav-plan` are tactic identifiers that have associated, respectively, the action that moves a robot, the action of moving a block on another one, the generation of a navigation plan, the execution of a navigation plan. Constants represent entities over which actions are executed.

Definition 3.1 (Primitive tactic) *Let ξ_i, with $i = 1, \ldots, n$, be either a constant, or a goal, or a fact or a variable. Let t_i be an n-ary tactic identifier. Then $(t_i \ \ \xi_1 \ldots \xi_n)$ is a primitive tactic.*

Thus, for instance, primitive tactics are

```
(goto room-100)
(put-block-on block-10 block-9)
(nav-plan-for (robot-at room-100))
(exec-nav-plan charge-hall-nav-plan)
```

where `block-10`, `block-9`, `room-100` and `charge-hall-nav-plan` are constants of the MRG language representing, respectively: two blocks in a blocks world, a location in the map of a building and a predefined navigation plan to reach a particular location (the hall of the building) from the area for charging batteries.

In the examples above, the level of abstraction of primitive tactics is very different; we have very simple actions in the external world linked to a primitive tactic (*e.g.* `put-on-block`) and very complex ones (*e.g.* `goto`). The level of abstraction depends on the application domain. Notice also that we have tactics that represent simple/complex actions in the real world (*e.g.* `put-on-block` and `goto`) and tactics that represent basic planning activities (*e.g.* `nav-plan-for` and `exec-nav-plan`).

Technically, MRG keeps track of the link between the tactic identifier and the action in a *tactic-action pair*, a data structure $\langle t_i, a \rangle$ where t_i is the tactic identifier of a primitive tactic and a a piece of code that can be executed. An action can contain parameters to be instantiated. For instance the module that is associated to the tactic identifier `goto` has a parameter that identifies the location the robot has to reach. MRG keeps track of the link between the constants of the language and the data structures that can instantiate action parameters in a *c-d pair*, a data structure $\langle c_i, d_i \rangle$ where c_i is a constant of the MRG language and d_i is the data structure a parameter of an action can be instantiated to.

The MRG tactic interpreter interprets primitive tactics by executing the corresponding actions. For instance, let $c_1, ..., c_n$ be constants and t_i a n-ary tactic identifier. Let a be an action, $\langle t_i, a \rangle$ a tactic-action pair, $d_1, ..., d_n$ n data structures and $\langle c_i, d_i \rangle$ n c-d pairs for $i = 1, ..., n$. The primitive tactic $(t_i \ \ c_1 \ldots c_n)$ is interpreted by executing a applied to $d_1, ..., d_n$. If the execution succeeds (returning the value v), then the interpretation of the tactic is said to succeed (returning v); if it fails, then the interpretation the tactic is said to fail.

3.2 Libraries of primitive tactics

Primitive tactics representing basic planning activities are organized in libraries according to the basic planning tasks that they implement. We describe below some tactic libraries to

interpreter. Tactics corresponding to goals implement a form of goal-driven reasoning, tactics corresponding to facts a form of data-driven reasoning. The task of the scheduler is very simple. It waits for a goal or a fact in the queue, then it activates a reasoner and it checks again the queue. Given a goal we have one and only one tactic that is executed. The reasoner does not check any precondition before activating a tactic. Neither the scheduler nor the reasoner contain any hardcoded problem solving strategy. Any reasoning is left to the tactics executed by the reasoners. This provides the capability (at least in principle) to respond in a very short time to any goal request and to any fact representing a stimulus from the external world. This should not mislead the reader. A tactic can include (as we will see in section 3 and 4) complex plan formation, execution and failure handling mechanisms. It can implement a whole planner. Then `MRG` has not a precompiled plan for each goal or fact. `MRG` provides the user with a tactic language that allows the user to define its own way to solve any goal or class of goals.
More than one reasoner can be active at the same time. This provides the capability to deal with asynchronous and parallel events and goals. In the current implementation, the interaction between asynchronous and parallel tactics is managed by primitive tactics that implement domain dependent communication mechanisms. A general management of parallel plan executions has still to be implemented in `MRG`.
The **knowledge bases** contain information about the external world, the internal `MRG` status and its resources. The system can have knowledge bases that have different representation techniques. They can be general purpose knowledge bases, for instance knowledge bases based on first order logic, and domain dependent knowledge bases, for instance a topological graph representing a navigation domain. Different tactics in the libraries access information represented with different formalisms in the knowledge bases.

3 The planning language

`MRG` provides a programming language designed to represent plans of actions and planning activities by means of tactics. Tactics are programs in the `MRG` language. Tactics are inductively defined on the base of *primitive tactics*. Primitive tactics are defined, among other things, by means of goals and facts. Goals and facts can be represented in a first order formalism. An expression of type *goal* may be `(robot-at room-100)`, where `robot-at` is a predicate and `room-100` is a constant. The same expression can be given a type *fact*, stating that the robot is actually at `room-100`. Primitive tactics are combined into *(compound) tactics* by means of a set of general combination operations called *tacticals*.

3.1 Primitive tactics

We call executable action types pieces of executable code, that, when executed, may fail or succeed. They include the usual notion of "actions in the external world". For instance, an executable action can be the module that moves a robot to a given location in a building or that moves a block on another one. But executable actions extend the usual notion, in the sense that they can be (implementations of) basic planning activities. For instance an executable action can be the module that, given a goal and a model of the world, generates a sequence of actions to be carried out by the robot (plan formation). An executable action can be the module that, given a plan, executes the corresponding actions (plan execution).
Executable actions are represented in `MRG` by means of *primitive tactics*. Primitive tactics

- plan generation is reasoning about a planning architecture, since generating a tactic means to build the composition of modules that implement the architecture;
- plan execution is running a planning architecture, since a tactic, when executed, activates the control mechanisms over the different modules of the architecture.

Then, rather than implementing a particular planning architecture, we have built MRG as a system that supports the representation, generation and execution of tactics. Different planning strategies are fully circumscribed in the tactics of MRG. Tactics can be fully customized. The user can chose the most suited primitive tactics from the libraries and build libraries of complex tactics thus defining the best planning architecture suited to the particular application. As a consequence, the architecture of MRG is relatively simple and completely general purpose. It provides the basic mechanisms to acquire user's goals, real world events, to activate external modules that execute actions in the real world and to perform tactic management.

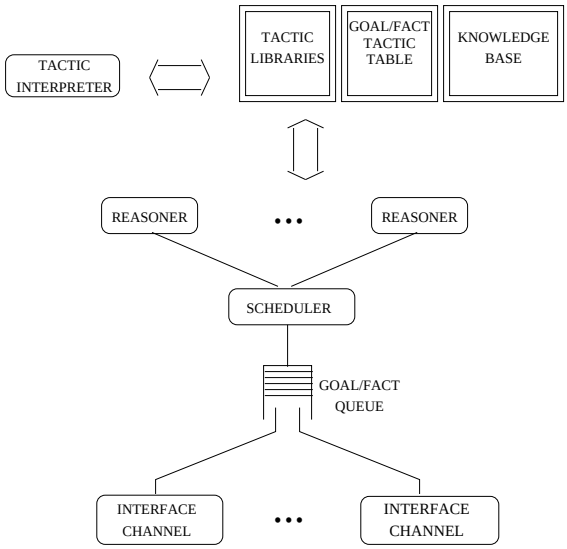

Figure 1: The architecture of MRG

The main components of the MRG system are the *interface channels*, the *goal/fact queue*, the *goal/fact-tactic table*, the *scheduler*, the *reasoners*, the *tactic libraries*, the *tactic interpreter* and the *knowledge bases* (see figure 1).

The **interface channels** provide the basic interaction between MRG and the external world. They give external goals and facts in input to MRG. Goals represent purposes to be achieved, like the goal to move a robot somewhere in a building. Facts represent information about the world, like the fact that the robot is actually in a particular location in the building. Goals and facts from interface channels are stored in a **goal/fact queue**. The queue can also contain goals and facts that are generated by MRG itself during the execution of a planning activity, for instance, the sub-goals generated during plan formation and the facts deduced by MRG.
The **scheduler**, as soon as a goal or a fact is added to the queue, activates a **reasoner**. The **reasoner**, given a goal or a fact, looks for the corresponding tactic in a **goal/fact-tactic table**, loads the tactic from the **tactic libraries** and executes it through the **tactic**

and basic planning activities of the planner are represented uniformly by primitive tactics.

3. Each control mechanism is represented by an extended notion of plan [Giunchiglia *et al.*, 1991], that we call **tactic**[1]. Intuitively, a tactic is a plan that composes primitive tactics in different ways.

   (a) The tactic building blocks are primitive tactics. A tactic is therefore a plan consisting not only of (representations of) the actions executable in the external world, but also of representations of basic planning activities.

   (b) Primitive tactics are composed into complex tactics by means of operators, called **tacticals**. Tacticals represent the basic control structures to build complex control mechanisms. They include sequences, conditionals, repetitions, structures to handle failure. A tactic is a plan that can describe when to generate a plan, when to respond to external stimuli and under which conditions to execute a generated plan. It can state whether it is better to replan a new course of actions after that a failure occurred or to execute directly some precompiled failure handling strategy.

As a consequence of these facts, within MRG the user can flexibly customize the whole planner to feature different behaviours depending on the requirements of complex, real world, large-scale applications.

An overview of MRG is given in section 2. Section 3 describes the language of tactics. A formal definition of primitive tactics and a description of the tactic libraries is given in section 3.1 and 3.2. Tactics and tacticals are defined in section 3.3. In section 3.4 we describe how they can be used to combine the basic planning activities and to extend the MRG libraries. In section 4 we show how in MRG we can implement and integrate several planning techniques and paradigms, from classical planning to deferred planning techniques and reactive systems. Section 5 gives a brief overview of how MRG has been used to build a complex real world application controlling a robot. Finally there are a discussion of some related work (section 6) and some conclusions and future developments (section 7).

2 An overview of the system

MRG is based on the idea that all the different control mechanisms and planning strategies must be "moved out" of the system implementation code of the planner [Spalazzi *et al.*, 1992]. Different planning architectures are represented by different plans, *i.e.* tactics. In MRG we have that:

- a plan represents a planning architecture, since
  - the different modules of the architecture are the different basic planning activities represented explicitly by primitive tactics;
  - the control over the activation of the modules are represented explicitly in tactics by means of tacticals;

[1] The term tactic is borrowed from ML [Gordon *et al.*, 1979], a procedural metalanguage for the specification of theorem proving strategies. There actually exist some similarities between the two languages, but there are rather strong conceptual and technical differences, mainly due to the different domains of application.

1 Introduction and motivations

The activity of most (general purpose) planning systems can be described in terms of few tasks, for instance reasoning to construct a plan solving a goal, *i.e. plan formation*; controlled execution of the real world actions described by a plan, *i.e. plan execution*; monitoring execution by acquiring information from the real world, *i.e. monitoring*; recovering from failures, *i.e. failure handling*. We call these tasks *basic planning tasks* and the (possibly different) algorithms that perform each of these tasks, the **basic planning activities** of the planner. We call the different criteria used by the planner to activate and to combine these tasks the **control mechanisms** of the planner.
Several basic planning activities and control mechanisms have been proposed so far. For instance, different plan formation activities are based on state space [Fikes and Nilsson, 1971] and partial plans [Wilkins, 1988] search; two different failure handling mechanisms are replanning [Wilkins, 1988] and execution of precompiled exception handling routines [Simmons, 1991b]. Different control mechanisms are implemented, for instance, by classical planners [Fikes and Nilsson, 1971; Wilkins, 1988], case-based planners [Hammond, 1990], conditional planners [Schoppers, 1987], deferred planning techniques [Durfee and Lesser, 1986], reactive systems [Brooks, 1986; Kaelbling, 1987].
Even though limited tuning operations are possible, most of the existing planning systems provide only one basic planning activity for a planning task and a fixed overall control mechanism. But real world and complex applications require that planning systems be customized flexibly according to the application domain.
First, most of these applications require that the system performs different basic planning activities in different situations. For instance, failure handling by executing precompiled exception handling routines is well suited to deal with emergency situations, while replanning seems better suited when the system needs to and has time to reason about the failure that occurred.
Second, different control mechanisms must be used and integrated in the same system. For instance, most real world systems need to generate a plan to anticipate predictable events and situations and, then, to execute the plan (as in classical planning). However, when information is not available, they need to interleave planning and execution (as in deferred planning techniques). In certain situations, the same application has not time to plan ahead, but it needs to respond immediately to environment changes (as in reactive systems).

We present a system, called `MRG`, that allows the user to build planners that perform different basic planning activities and control mechanisms depending on the situation and on the application domain. This is achieved through some main features that make `MRG` different from any system proposed so far.

1. Each basic planning task is organized in a library that includes different basic planning activities. `MRG` has libraries with different plan formation, plan execution, real world information acquisition and failure handling mechanisms.

2. Each basic planning activity is represented explicitly as an expression of a planning language. We call these expressions **primitive tactics**. A primitive tactic can describe different ways of generating a plan for a given goal, responding to external stimuli, executing a generated plan and acquiring information from the external environment. Primitive tactics can also represent actions executable in the external world, like moving a block, moving a robot to a certain position and so on. Actions in the external world

Abstract

We are interested in the development of planners that work in real-world and large-scale applications. Most of these applications require systems that perform and combine the usual planning tasks, *e.g.* plan formation, plan execution, monitoring and recovering from failure, in a problem dependent way. Furthermore, in most cases, these systems must integrate the various different planning techniques proposed in the literature, *e.g.* planning ahead, deferred and reactive planning. In this paper we present a system (called `MRG`) that eases all these difficulties by representing explicitly all the planning tasks and by combining them using a small set of powerful control structures. At the moment, we are experimenting with `MRG` in a complex large-scale application that is briefly described in this paper.

MRG: building planners for real world complex applications*

Paolo Traverso[1]
Alessandro Cimatti[1]
Luca Spalazzi[2]
Enrico Giunchiglia[3]
Alessandro Armando[3]

[1]IRST - Istituto per la Ricerca Scientifica e Tecnologica, 38050 Povo, Trento, Italy
phone: +39 461 814 440
[2]Istituto di Informatica, University of Ancona, Via Brecce Bianche, 60131 Ancona, Italy
phone: +39 71 220 4832
[3]DIST, University of Genova, via Opera Pia 11/A, 16145 Genova, Italy
phone: +39 10 353 2983
e-mail: leaf@irst.it cx@irst.it spalazzi@irst.it enrico@dist.unige.it armando@dist.unige.it

*This work has been done as part of MAIA, the integrated AI project under development at IRST. Partial support has been provided by CNR (Italian National Research Council), Progetto Finalizzato Robotica (Special Project on Robotics). Fausto Giunchiglia has supervised the development of the whole project and provided many of the underlying intuitions. Sandro Dalbosco, Roberto Giuri and Marco Marinucci have implemented part of the application developed in `MRG`. Some material in this paper is contained in the reports No 9112-19 ("Programming planners with flexible architectures"), No 9204-05 ("Beyond the single planning paradigm") and No 9205-11 ("Navigation by combining reactivity and planning").

ISTITUTO PER LA RICERCA SCIENTIFICA E TECNOLOGICA
I 38100 TRENTO – LOC. PANTÉ DI POVO – TEL. 0461–314444
TELEX 400874 ITCRST – TELEFAX 0461–302040

MRG: BUILDING PLANNERS FOR REAL WORLD COMPLEX APPLICATIONS

P. Traverso, A. Cimatti, L. Spalazzi, E. Giunchiglia, A. Armando

February 1993
Technical Report # 9302-21

To appear in *Applied Artificial Intelligence*, Vol. 8, No. 3, 1994.

ISTITUTO TRENTINO DI CULTURA