A Software Component for Simultaneous Plan Execution and Adaptation

Sylvain Joyeux, Rachid Alami, Simon Lacroix

▶ **To cite this version:**

Sylvain Joyeux, Rachid Alami, Simon Lacroix. A Software Component for Simultaneous Plan Execution and Adaptation. IEEE Conference on Intelligent Robots and Systems, Oct 2007, San Diego, United States. hal-00166797

HAL Id: hal-00166797
https://hal.science/hal-00166797

Submitted on 10 Aug 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Software Component for Simultaneous Plan Execution and Adaptation

Sylvain Joyeux, Rachid Alami and Simon Lacroix
LAAS/CNRS, University of Toulouse, Toulouse, France
firstname.lastname@laas.fr

***Abstract*— This paper presents a software component, the plan database, which provides the needed services to define plans, execute them and more importantly adapt them during execution. This plan database handles fully dynamic plans (insertion and removal of tasks), defines task transformation operators and provides tools for safe concurrent execution and modification of plans. These features are essential in multi-robot and human-robot contexts, where tasks need to be easily passed between systems and plan adaptation helps coping with the unpredictability inherent to systems where multiple agent make decisions.**

I. INTRODUCTION

In robotic systems, the planner rarely produces results directly executable by the platform. It is often needed to have an intermediate component, whose role is to manage the functional layer based on the execution state and the information available in the plan. Moreover, this execution component can handle “simple cases”, so that upper layers do not have to manage all events that come from the functional layer: it typically performs error recovery and a limited form of script-based plan generation. To handle the complexity of these executives, tools like TCA [1], TDL [2], OpenPRS [3] or ESL [4] have been designed.

One problem with this approach is that two components, the planner and the executive, are keeping two different plans. The first often lacks information about the execution state, while the second lacks information about high level goals, which are usually handled by higher layers. This makes it difficult to deploy global plan analysis tools: state and time estimation systems can greatly benefit of a complete view of all processes that are running in the system.

On the contrary, in the Claraty [5] architecture, a single plan is being managed by a central component. Claraty then provides a simple mechanism to ensure that the part of the plan being executed will not be changed by the planner: a floating “line” separates the long-term plan, which can be freely modified by the planner, from the short-term which is read-only for the planner, and is exclusively handled by the executive.

The IDEA [6] architecture defines a hierarchy of agents, each of which plans a subset of the whole plan and then sends part of it to other agents for further processing. This allows for instance to use a fast, reactive planner for low-level tasks and a long-term planner for high-level ones. The IDEA architecture has been designed so that consistency is maintained in the plans of all agents. However, for multi-robot systems, this kind of decomposition has the limitation that since no agent has a global view of the robot plan, it is difficult to detect redundancy between robots.

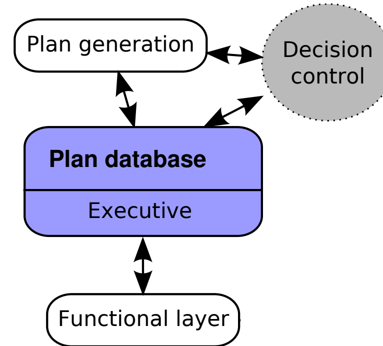

Fig. 1. The plan database is a system component that represents and maintains plans during execution

All these architectures lack the ability to efficiently remove tasks from plans. The reason is often that the planners themselves lack this ability. Our experience in developing robot systems show that this capability is essential in a number of contexts, *e.g.* in multi-robot mechanisms: tasks may be transferred back and forth between robots, for instance to balance robot workload, or to replace one robot’s task by another’s to remove redundancies.

This paper introduces a plan management component, the plan database (pDB), which provides services aimed at addressing these issues:

- fully dynamic: it defines a set of plan modification operations that are to be performed by the pDB.
- simultaneous execution and modification of the plan. It provides a generic tool to handle conflicts between plan modifications and execution.
- reduce redundancy: plan operators promoting the reuse of tasks already present in the plan

We first provide an overview of our system, and present its behavior using an example. Then, we describe how plans are modeled in the pDB and how they are managed during execution. Finally, we describe the current implementation of this system and outline future work.

II. OVERVIEW

Fig. 1 presents the context in which the pDB is embedded. Plan generation tools (planners) are responsible for producing consistent plans, and the functional layer is a service layer which provides algorithms and interfaces between the software and the real world. Between these two, we introduce

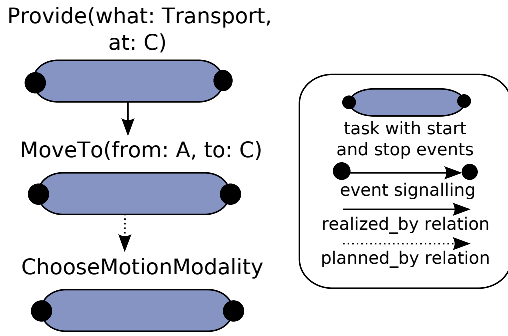

Fig. 2. Initial refinement of the `Provide` task

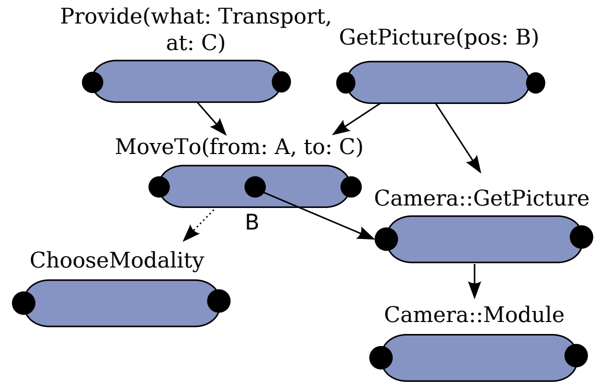

Fig. 3. The rover now has two goals: going to C and taking a picture at an intermediate point B. The corresponding plan is a directed graph of tasks and events.

three components: the pDB maintains a plan, which is a graph of tasks and events defining what the robot may do in the future and how it will do it. This plan is continuously interpreted by the *executive* to produce actual actions, using an event-based model. The *decision control* component has two roles: first, it may call the planners for instance to adapt the plan for new missions or for contingency planning. Second, it is called during execution to handle the choices that have to be made: since our software system allows simultaneous planning and execution, we may need to choose between the main plan being executed and the partial plans that are being built by the plan generation tools. This other role of the decision control component is explained in more details with the central tool for simultaneous planning and execution: *transactions* (Sec. IV-D).

Let's assume that an operator needs a transport to a given point C. A robot is given this particular mission, which is an initial plan made of only one task: `Provide(what: Transport, at: C)`. This plan cannot be executed yet since there is no information on *how* this would be done. We therefore need to develop this task.

From the database point of view, planner results are a set of plan modifications which, when applied to the current plan, will produce a new global and consistent plan. The database does not check the consistency of generated plans, it provides tools to adapt them and to safely manipulate the global plan while concurrently executing it.

Since planning is in general far from being fast, the system must be able to ensure that a set of modifications is valid despite the changes brought by execution. To that end, the database provides *transactions*[1]: this is a sandbox in which planners can freely generate the needed set of modifications *without changing the global plan*, this set being applied atomically when planning is finished. These modifications are gathered in one well-defined object such that the pDB can check their validity with respect to the global plan being executed.

In our example, a planner is selected for `Provide` by the decision control. This planner begins a transaction, produces the set of plan modifications leading to the plan of Fig. 2 and applies them by *committing* this transaction. Before it can be executed, however, a specific `MoveTo` modality has to be chosen. For example, our all-terrain rover has two motion modalities [7]. The NDD modality performs reactive obstacle avoidance by using a 2D laser range finder and can be used only on flat terrain. For more difficult terrain, the P3D modality uses stereovision to produce a 3D terrain model, which is then used to compute local robot path. The first one allows faster movement, while the second one is more robust. We therefore want to use NDD when the terrain allows it, and P3D otherwise.

[1] which are an adaptation of the transactions found in "classical" databases

Choosing between these two modalities is done by the `ChooseModality` task, which *plans* the `MoveTo` task. Note that the plan database knows that it can switch between `NDD::MoveTo` and `P3D::MoveTo` in this context, since from the point of view of `Provide`, they are both equivalent to `MoveTo`. The *replace* operator handles the exchange between such alternatives.

However, changing the kind of `MoveTo` task on the fly is not that simple: we cannot have both modalities active at the same time, as it would mean that there is a way to send commands to engines from two different sources, which is forbidden in our functional layer. Therefore, when we switch motion modality there is a period of time where none is active, which breaks the plan. However, if the duration of the switch is short compared to the system dynamics, this break can be allowed. The pDB defines *plan repairs* to do this in a controlled way.

Suppose that during the execution of `Provide`, the operator decides that a picture is needed at an intermediate point B, which is not far from the planned path. The `GetPicture(at: B)` mission is inserted into the rover plan using a transaction. To allow synchronization between the movement and `GetPicture`, a new *event* B is dynamically added to `MoveTo` (Fig. 3). Adding this event to the task instance allows to express that we need synchronization on a particular position of the robot and that the robot should move close to B (events are presented in more details in III-A). If that is no longer possible, the B event enters a particular state which allows us to determine that the `GetPicture` cannot be executed anymore.

If the motion reaches B before the plan change has been applied to the plan, the transaction cannot be committed

anymore: it depends on parts of the plan that lie in the past. Thus, the transaction becomes *invalid*. In our case, this can be solved by discarding the transaction, which abandons `GetPicture`.

Having introduced the main pDB concepts, we can define its execution cycle as follows:

- event gathering and propagation: get the set of external events we are listening to and react to them according to the information in the plan.
- check plan structure and propagate the corresponding exceptions: the plan structure implies some constraints on tasks. Failures to meet those constraints are detected and can be recovered at this stage.
- check plan structure and kill the tasks involved in the remaining exceptions: we do a second check on the structure. The errors not repaired in the previous step are dealt with by killing the involved activities.
- get the set of tasks that are not required by any robot mission and remove them: the pDB knows which activities are useful for the robot missions and which are not. The tasks that are not useful are killed.

III. PLAN OBJECTS

Plans are graphs of two kind of objects: events and tasks (Fig. 3). Events describe singular happenings during task execution and the event graph encodes what action should be taken when an event is achieved. Similarly to TDL and other systems, tasks describe processes and the task relations describe the interactions between these processes.

A. Event graph

Events are achievements in the system, for instance "changed speed" or "is at position P". During the execution, an event can have the following states:

- *achieved* or *emitted:* the event has occurred.
- *achievable:* it is possible that the event will be achieved in the future. This handles cases where parts of the plan won't be executed because they depend on the achievement of some events, but these events will never be achieved.

Moreover, an event is *controllable* if there is a way to make sure that it will be achieved. For instance, a $e_{robot_stopped}$ event is controllable, since the system knows how to stop the robot. On the contrary, a $e_{touched_obstacle}$ event is not. Controllability is implemented by attaching a procedure, called the *event command*, to an event. The event model demands that if an event command is called, then this event will be achieved at some point in the future. An event is *pending* if its command has been called, but the event is not achieved yet.

Events are linked with directed relations, which define what action to take when a particular event is achieved. Two event relations are defined:

- *signaling.* if e_1 signals e_2, then the command of e_2 is called when e_1 is achieved. e_2 must be controllable.
- *forwarding.* if e_1 is forwarded to e_2, then e_2 will be achieved when e_1 is achieved. e_2 can be contingent.

B. Task models

While controllable events represent a deterministic link between an achievement and its command, tasks represent processes where simply calling the command (starting the task), does not allow to predict that it will successfully fulfill its purpose. In the plan database, *task models* are defined as a set of events, which are the milestones of the task execution. Plans are then made of *task instances*, which are defined by a task model and a set of parameter values. During execution, achievement of the task events is determined either internally by the task itself, or externally by forwarding external events. Since a task can define multiple events, which can be mutually exclusive or not, our plan model defines multiple execution paths. Unlike conditional plans, however, they are not necessarily mutually exclusive.

All task models define the following events:

- $e_{started}$: the task is started.
- $e_{aborted}$: the task execution support (UNIX process, hardware) has terminated unexpectedly. In case of tasks that control hardware, it means that the state of the underlying hardware is unknown. Handling of execution support will not be presented in detail in this paper.
- e_{failed}: the task has terminated, but did not realize its purpose.
- $e_{success}$: the task has realized its purpose. A task is *achieved* when its $e_{success}$ event is.
- $e_{stopped}$: the task has terminated. If this even is controllable, the task is interruptible.

Obviously, no task event can be achieved before $e_{started}$ or after $e_{stopped}$, and all of above events can only be achieved once for a given task instance. Note that all events except $e_{started}$ describe a termination of the task. We therefore would want that, for instance, if e_{failed} is achieved then $e_{stopped}$ is achieved too. Forwarding $e_{aborted}$ to e_{failed}, and forwarding e_{failed} and $e_{success}$ to $e_{stopped}$ ensures that.

Task models are managed in hierarchies, where a parent model defines a more generic task than a child model. This hierarchy is different hierarchy than the refinement hierarchy of task instances presented in Sec. III-C.

- as tasks are defined by a set of parameters, a child model defines at least the same parameters as its parent. It can define more, but not less.
- a child model defines a super-set of the events defined in its parent model

It follows that a task t_1 can be substituted by another task t_2 if its model is a child of the model of t_1 and if its parameter set is included in t_1's parameter set.

For instance, `NDD::MoveTo` and `P3D::MoveTo` are submodels of `MoveTo` (Fig. 4), which takes two arguments: the origin and the destination. Submodels of `MoveTo` can take more arguments, such as specific parameters for the algorithms of each modality. Moreover, the system can dynamically add a new event B on `MoveTo` instances, which is achieved when the robot crosses a geometric point B. If new events have been added, then specialized task instances can be used for substitution only if the same events can be added to them.

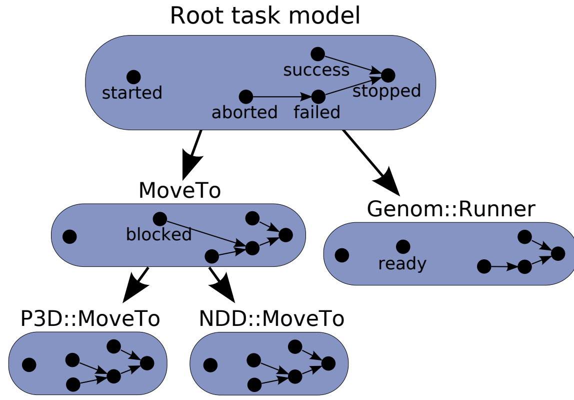

Fig. 4. Examples of task model hierarchy. The abstract `MoveTo` task is implemented by more specialized models. Another unrelated model, `Genom::Process`, is used to represent the Unix process of the modules in our functional layer.

C. Task instances graph

To form a plan, we need to describe task interactions. The core model defines two kind of relations between tasks, which form directed acyclic graphs: $realized_by$ and $planned_by$. We use the convention that, if a task t_1 is the parent of another task t_2 in any relation graph, then t_2 is directly useful for t_1. This is used to determine what task is useful to the robot missions.

1) Task refinement hierarchy: Task refinement is represented by the $realized_by$ relation: a task T is $realized_by$ a task t if the achievement of T requires the achievement of some of t's events. In that case, t is a child of T and T is a parent task of t. The *child tree* is the tree formed by the $realized_by$ relation and rooted by T. Unlike TDL or TCA, however, the tasks do not form a task tree but a *task graph*: one task can have many children and many parents, which allows to easily reuse tasks already in the plan.

The $realized_by$ relation has the following parameters, which are needed for plan management:

- a $(model, arguments)$ pair which defines what kind of task T needs, following the substitution principle explained earlier.
- a set E_f of failure events from the child task. If any of these events is achieved, we know that the child task will not realize what the parent task was expecting.
- a set E_s of success events from the child task. If any of these events is achieved, then the child task performed what the parent was expecting.

Usually, $E_s = \{e_{success}\}$ and $E_f = \{e_{failed}\}$. In Fig. 3, `GetPicture` needs the `MoveTo` task only until B is achieved, thus E_s is $\{e_B\}$. Note that the achievable state of the events in E_s is important.

2) Planning tasks: t is planned by T if T represents the planning process in charge of the achievement of t. In general, it means that all or part of the subtree of t has been produced, and can be changed by T. Planning tasks can also be used to represent situations where a set of tasks are abstract (not executable), and will be developed later. It also allows to integrate repair capabilities of planners, as the planning tasks are involved in exception handling (see IV-C)

On Fig 3, we see that `MoveTo` is $planned_by$ the `ChooseModality` task, whose role is to choose continuously between the two motion modalities, either in proactive manner or because the current modality has failed. The latter case is presented in section IV-C.

IV. PLAN MANAGEMENT

The pDB defines the following services for plan management:

- task replacement
- automatic plan cleanup mechanism, to remove unneeded tasks
- handling of errors using both a conditional plan structure and exceptions
- transactions, a tool to concurrently modify and execute the plan

A. Task replacement

The $replace(t, T)$ operator transparently replaces the task t in the plan with a new task T. This is only possible if T is a valid substitution of t for all the hierarchy relations t is part of. To replace running tasks, we need to check that the two task models are equivalent (which is done by the pDB), but also that they are in the same *execution state*. This is done internally by the two tasks: T should provide a method which ensures that its state is equivalent to the one of t, given the task models needed by t's parents. In simple cases like the `MoveTo` replacement, it simply ensures that a running task is replaced by a running task.

B. Plan cleanup

During the lifetime of the system, tasks will be inserted and removed dynamically. We need to be able to determine what tasks can be removed, given that we know what are the high-level goals of the robot: if we remove a high level task like `GetPicture` in Fig. 3, we need to remove *some* of its children (in this case, `Camera::TakePicture`). There are therefore two kinds of tasks in a plan: the *missions* are the high-level goals of the robot and the remaining of the plan exists to achieve these missions. It follows that the plan can be split into a set of tasks needed by the set of missions, and a set of tasks that are not useful in the context of the current missions. The first set is part of the child tree of a mission, while the second is not. Note that a mission is in general not explicitly removed by an external tool: it is marked as not being a mission anymore, and the system will clean it up itself.

It is impossible to remove tasks that have parents (in any of the task graphs): it is possible that they take part in the parent task's shutdown process. Therefore, the cleanup algorithm is as follows:

1) remove all tasks in the plan which are not needed anymore, which have no parent task and which are not running. Loop until there is none to remove.

Program 1 Definition of an exception handler in the `ChooseModality` task model. The change_modality method creates the new planning task and returns it.

```
class ChooseModality
  on_exception(ChildFailedError) do |error|
    if error.task == planned_task
      new_planner = change_modality
      plan.repair(new_planner, error)
    end
  end
end
```

2) compute the set of tasks which are not needed anymore and which have no parent. All tasks in this set are running because of the previous step.
3) for all tasks in this set, call $e_{stopped}$ if it is controllable.

C. Error handling

We want to be able to express plans where failure is explicitly taken into account: like ESL [4], and unlike TDL [2], we do not consider that failures are always exceptional conditions. While TDL would for instance handle a failed movement as an exception [8], this is an error so common in unknown environments than its correction should have already been planned.

To that end, *plan repairs* are tasks that are handling some non-nominal events (separation between different failure modes is done by defining new events which are forwarded to e_{failed}). Moreover, repairing a plan can take time, and it may be perfectly fine to keep the plan broken for a period of time short with respect to the system dynamics: we associate a repair with a timeout.

However, if the failure event is not handled – or if the repair fails – an *exception* is raised. The system then passes the exception up in the hierarchy graph, checking at each task if the task or its planning task both defines an exception handler and that this handler accepts the exception (and thus repairs the plan). The following two rules are added to handle parallel branches in the task graph:

1) the same exception shall be merged when two parallel branches meet in the task hierarchy.
2) if two handlers are found in parallel branches for the same error, we let the decision control decide which of the two repairs should be applied to the plan.

For instance, the `ChooseModality` introduced in Sec. III-C.2 can define an exception handler for the `ChildFailedError` exception (Prog. 1), which is raised if a child task failure is not handled by the plan. This handler can trigger replanning of `MoveTo` and thus repair the plan. The planning of the new modality is done asynchronously, and the exception handler defines the planning task as a plan repair.

D. Concurrent plan modification and execution

While one can modify the global plan directly, it is dangerous as the executive could begin the execution of some part of the plan that is not finished yet. Currently, most executives solve this problem by forbidding the modification of the short-term plan. We find this solution quite limited since a low-level executive must often do reactive modifications. Our pDB offers *transactions*, which gives a context to build a set of plan modifications outside of the global plan, modifications which are then committed atomically (either all at once, or not at all) and synchronously (they have their own slot in the execution cycle). Therefore, the use of transactions guarantees that the plan being executed is always sound as long as the generated plans are.

As we saw in section II, concurrent execution and planning can lead to *invalid* transactions: in our example, the B event used in the transaction is emitted by the execution. Therefore, the plan represented by the transaction becomes invalid and the transaction cannot be committed. An invalid transaction can be discarded or repaired by changing the transaction, the global plan, or both. Choosing the way to handle this invalidation is done by the decision control component, whose role here is two choose between a modification of the transaction or a modification of the plan, by calling plan generation components if needed. Moreover, if an execution operation (like a signal) invalidates a transaction, the decision control component shall choose between this operation and the transaction invalidation. If the signal is chosen, the transaction is invalidated and the resolution procedure is the same as before. However, if the transaction is chosen, an exception is thrown from the signal source, whose effect would be to either change the plan so that the `MoveTo` task shall not be stopped, or remove the part of the plan which was relying on this signal.

V. IMPLEMENTATION AND FUTURE WORK

The pDB is currently implemented in the Ruby language. We use the object-oriented capabilities of Ruby as a way to define task models and task instances as classes (and class hierarchies) and objects (Prog. 2), thus gaining a lot of time in the development of the prototype. Moreover, developing the system in a general-purpose language promotes code reuse in supervisors: the development for our rover shows that a great level of modularity can be achieved by defining generic task models which are subclassed by specific ones and because it allows to extend the system by defining libraries of often-used plan modification operators.

The GenoM [9] functional layer provides the functional modules on top of which the supervisor is built. These modules are integrated seamlessly in the supervisor by defining a task for each GenoM activity (an activity is a request which is being handled by the module). The UNIX process of the GenoM module is also represented, and all GenoM requests depend on it: the termination of a GenoM module is handled at the plan level by emitting the $e_{aborted}$ event on the corresponding requests, which will trigger pDB's error handling.

Fig. 5 is a partial representation of the generated plan. We see `PlanningTask` and `PlanningLoop` tasks which represent the handling of plan generation processes:

Program 2 Code example to define (left) the generic task model and (right) the abstract `MoveTo` model. In order, `MoveTo` has two arguments, adds the terminal event $e_{blocked}$, makes e_{failed} and e_{stop} controllable. e_{stop} commands calls e_{failed}'s command. e_{failed} does nothing: we rely here on the pDB's cleanup mechanisms to kill its children when it is finished. If a specific cleanup order was needed, the event command would have defined a chain of events to define it.

```
class Task
  event :start
  event :failed
  event :success
  event :stop
  on :failed=>:stop
  on :success=>:stop
end
```

```
class MoveTo < Task
  arguments :from, :to
  event :blocked
  on :blocked=>:failed
  event failed do emit(:failed) end
  event stop do failed! end
end
```

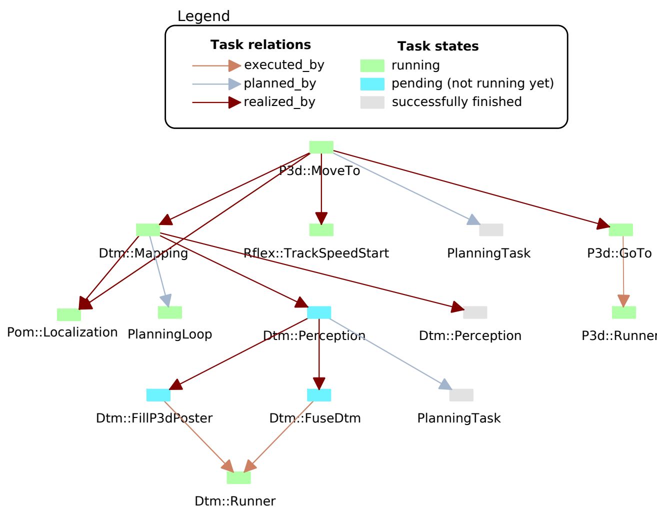

Fig. 5. Partial view of the task structure executed for the Dala rover. The `Runner` task represent the Unix processes of the Genom modules which execute requests like for instance `Bitmap::FuseLCLMap`.. The plan-generation tasks `PlanningTask` and `PlanningLoop` are briefly described in section V

- `PlanningTask` represents an asynchronous plan-generation process which uses the simple script-based plan generation.
- `PlanningLoop` develops and synchronizes sequences of the same action. In this plan, the terrain mapping `Dtm::Mapping` is realized by a sequence of perceptions (`Dtm::Perception`) executed in a loop.

The algorithms that are used during the cycle execution are either O(1) or O(N) where N is the number of tasks in the system. More CPU intensive tasks are done asynchronously in separate threads thanks to transactions. It follows that the current implementation is quite efficient, regardless of the slowness of the Ruby interpreter itself: the current execution cycle for the rover supervisor is below 10ms. The downside of using Ruby is the slowness of its garbage collector. It is not uncommon that the interpreter blocks for 40-50ms because of Ruby's garbage collection. However, the development of incremental garbage collectors in for instance the Java virtual machine shows that it is possible to keep this timespan controlled. We therefore run the controller at a cycle length of 100ms, which is enough for our needs and keeps the garbage collector issue under the cycle length.

One of our research objectives is to allow the simultaneous use of multiple planners in the same system, in order to use the more efficient planner according to current needs. This cannot be achieved without plan merging mechanisms. The development of this $merge$ operator is an ongoing work that is based on the presented substitution principles, event and task structure and an additional conflict predicate for tasks that can not be executed in parallel.

Finally, the pDB would benefit greatly of an estimation of the time of event achievements. Since we represent all processes in the system, including planning, we could develop advanced scheduling schemes to balance between planners needing the most precise information available (plan as late as possible) and the executive needing the resulting plan (make sure that the plan is ready when execution begins). The information needed for that will be given by the task models: like Claraty, the pDB will expect task instances to give information about event achievement, using internal models that are not known to the pDB engine. These information will then be used to compute a complete temporal model by using event and task structures.

VI. CONCLUSION

The contributions of this paper are the plan model and the set of plan management mechanisms built around it. The pDB allows insertion and removal of tasks, using the task hierarchy to detect the consequences of the removal operations. It also provides a mechanism to safely modify a plan online, *transactions*. By the modular nature of the system, we expect that the pDB can be extended and used in various contexts, including multi-robot systems and human-robot interaction.

REFERENCES

[1] R. Simmons, "Concurrent planning and execution for autonomous robots," *IEEE Control Systems Magazine*, vol. 12, 1992.

[2] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Proceedings of IEEE IROS*, 1998.

[3] F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A high level supervision and control language for autonomous mobile robots," in *Proceedings of ICRA*, 1996.

[4] E. Gat, "ESL: a language for supporting robust plan execution in embedded autonomous agents," in *Proceedings of the IEEE Aerospace Conference*, 1997.

[5] T. Estlin, R. Volpe, I. A. D. Nesnas, D. Mutz, F. Fisher, B. Engelhardt, and S. Chien, "Decision-making in a robotic architecture for autonomy," in *Proceedings of 6th i-SAIRAS*, 2001.

[6] A. Finzi, F. Ingrand, and N. Muscettola, "Robot action planning and execution control," in *Proceedings of IWPSS*, 2004.

[7] T. Peynot and S. Lacroix, "A probabilistic framework to monitor a multi-mode outdoor robot," in *Proceedings of IEEE IROS*, 2005.

[8] R. Simmons and E. Coste-Manière, "Architecture, the backbone of robotics systems," in *Proceedings of ICRA*, 2000.

[9] S. Fleury, M. Herrb, and R. Chatila, "Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture," in *Proceedings of IROS*, 1997.