The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Becoming Action-aware through Reasoning about Logged Plan Execution Traces

Lorenz Mösenlechner, Nikolaus Demmel, Michael Beetz

Intelligent Autonomous Systems Group
Department of Informatics
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
{moesenle,demmeln,beetz}@cs.tum.edu

Abstract— **Robots that know what they are doing can solve their tasks more reliably, flexibly, and efficiently. They can even explain what they were doing, how and why. In this paper we describe a system that not only is capable of executing flexible and reliable plans on a robotic platform but can also explain control decisions and the reason for specific actions, diagnose the cause of failures and answer queries about the robot's beliefs. For instance, when queried why it opened the cupboard door, the robot might answer that it did so because it believed Michael's cup to be in there. This type of reasoning is not only helpful for debugging but also provides the mechanisms for complex monitoring and failure handling that is not based on local failures and exception handling but on the expressive formulation of error patterns in first order logics. Our system is based on semantic annotations of plans, a fast logging mechanism and the computation of predicates in a first-order representation based on the execution trace.**

I. Introduction

Robots that know what they are doing can solve their tasks much more competently, reliably, and flexibly and can even explain what they were doing, how and why. An action-aware robot should be capable of answering queries with respect to what it was doing in the following ways. For example, after a table setting activity the robot answers the query about why it did not put butter from the table into fridge with that when it completed the cleaning the table task, it could not perceive any more items lying on the table. This is not only an interesting feature for debugging control programs but allows for sophisticated handling of unwanted situations that are far more complex than simple execution failures such as dropping an object.

To give answers like the ones above, the robot must know the structure of its plans, the intentions and roles of subplans, it must infer its task relevant beliefs at given stages of plan execution and remember the success and failures of subplans.

In this paper we describe an extension of a plan-based robot control system CRAM (Cognitive Robotic Abstract Machine) [1] that realizes just this functionality. CRAM is a framework for the development of cognitive robot control software, including robot control, reasoning and belief state management. Among others, it contains a plan language that has been designed in particular for the domain of mobile robotics. It contains special language constructs for concurrency, task synchronization and program variables that reflect the state of the world. We extended CRAM to log the execution of plans, including the values of plan parameters, the activation and deactivation of subplans, their outcomes, and the beliefs of the robot in the course of the episode. Our extensions not only include logging, but also a first-order representation of the logged data which allows for the formal specification of the questions we want to answer and for making inferences on the logged data structures to answer these questions.

II. Related Work

In this paper we describe a system that can reflect itself, i.e. that can answer questions about the internal data structures, the course of action and the decision making process. The only similar system the authors know about that runs fully integrated on a robot platform and is able to reflect about itself is the GRACE system [2] which gave a talk about itself at a conference on artificial intelligence. This included information about it's capabilities and the tasks being executed.

Similar systems in the area of artificial intelligence are meta level architectures, such as a Prolog interpreter implemented with Prolog clauses [3]. In this approach, programs are interpreted as data that can be reasoned about. Thus, aspects of the interpretation are made explicit. But in contrast to these systems that are applied on disembodied problems inside a more or less static problem domain specified as facts in knowledge bases, our system allows to reason about plans that are executed on a robotic platform that acts in a complex and dynamic environment. Our system allows to handle uncertain knowledge and beliefs, and grounds the knowledge representation in the underlying data structures of the robot.

The work described in this paper is based on an enhanced version of the declarative and expressive plan language described in [4]. There, the authors develop a plan language for the definition of complex robot behavior. The constraints for plan design, in particular the specification of declarative goals indicating the purpose of code parts, have been shown in [5]. Besides the modeling of navigation tasks, our system

scales with respect to reasoning about perception (based on computer vision), the relation between objects and their representation in the robot's belief, as well as reasoning about complex manipulation tasks.

The first order representation described in this paper can be compared to Temporal Action Logics (TAL) [6]. Whereas TAL's authors define a language for reasoning about action and change, we add a concept of intention. That means, we cannot only describe what happened but also what the robot wanted to achieve.

III. System Overview

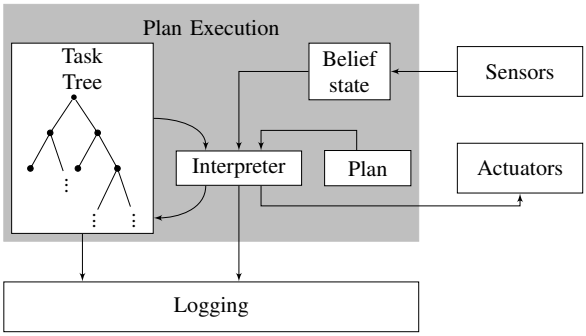

Fig. 1. Plan execution

CRAM provides a framework to write robot control programs that can not only be executed but are also transparent in the sense that they can be reasoned about. To answer questions such as "why didn't you pick up the cup on the table?", such a system must provide 1) the ability to record all important aspects of program execution and 2) mechanisms to turn the recorded data into a representation that allows to make inferences, i.e. a representation in first-order logic.

Figure 1 gives an overview of the execution component of CRAM. When CRAM gets a task to achieve, it first selects a suitable plan from a plan library and starts executing it. A plan in CRAM is a concurrent, reactive, perception guided control program that is carefully designed to allow for inferring the purpose of control routines, to make inferences about intentions, the belief state of the robot and errors occurring during plan execution. It contains explicit annotations in the form of special statements that assert the semantics of the plan in first-order logic. For instance, a plan that places an object *Obj* at Location *Table* is annotated by *Achieve(Loc(Obj, Table))* and one of its sub-plans is *Achieve(ObjectInHand(Obj))*.

Plan execution incrementally creates and updates a task tree that holds the stack frames of the control routines that are generated during plan execution. Sensor measurements and actions in the world continuously update the belief state. This information is recorded by a logging component that allows to reconstruct the complete state of program execution at any point in time.

In order to answer questions about the reason for failures of tasks, the purpose of tasks or which actions have been performed and why, a first-order representation of logged episodes is created. This includes predicates for accessing the task tree, for querying the stack frames, for investigating the semantic annotations, the belief state and percepts. For instance, to assert the status of a task at time t, we use the term

$$Holds(TaskStatus(tsk, status), t)$$

In this section we will show how plans are represented and executed and how the belief state is represented and accessed in the control program.

A. Plan execution

The execution of a plan is completely (but not necessarily deterministically) determined by the program state: the program counter and the variable values. In program execution these data are usually kept in a stack of task frames. Thus, everything that the robot "believes" in to decide on the course of action is at sometime somewhere on its execution stack. An example of a stack frame, which we call a *task* data structure is depicted in Figure 2. The task data structure contains the following data. The task environment contains the variables in the scope of the task, their values at different stages of execution, and the state of plan execution when these values were assigned. Thus the local variable *OBS* was initialized to *()* and then set to the set of object descriptions *(DES-17 DES-18)* at the end of task *t-7*. The task status contains the change of the task status during plan execution and when the status changed.

TASK *T-6*

SUPERTASK	T-4			
TASK-EXPR	(ACHIEVE (OBJECT-CARRIED DES-17))			
TASK-CODE-PATH	((POLICY-PRIMARY) (STEP-NODE 1) (COMMAND 1) (STEP 2))			
TASK-ENVIRONMENT	OBS	(BEGIN-TASK T-4)	()	
		(END-TASK T-7)	(DES-17 DES-18)	
TASK-STATUS	TI10	CREATED	TI10	ACTIVE
	TI13	DONE		

Fig. 2. Conceptual view of an executed task.

The location of the task in the task tree is denoted by a unique path, also saved in the task structure, as well as the executed expression.

B. Plan Representation

In order to reason about logged execution scenarios, plans must be written in a way such that the reasoning system can infer the purpose of a certain plan part. We annotate control routines by first-order expressions that state their purpose. The annotations are descriptions of the actions and the states in the world that these actions operate on. That includes *achieving* states and *perceiving* them. The states that the actions operate on are called occasions. More specifically, occasions are states of the world that hold over time intervals during program execution. By naming a control routine *Achieve(s)*, the programmer asserts that the purpose of the routine is to achieve state s, i.e. the corresponding occasion. Thus, if there is a control routine *Achieve(s)* on the execution stack the system infers that the robot currently has the intention to achieve state s. A routine to achieve state s that terminates with a successful status implies that the

robot beliefs that the occasion s now holds. On the other hand, when the control flow reaches an achieve statement that already holds, the control routine succeeds immediately because the occasion does not need to be achieved anymore.

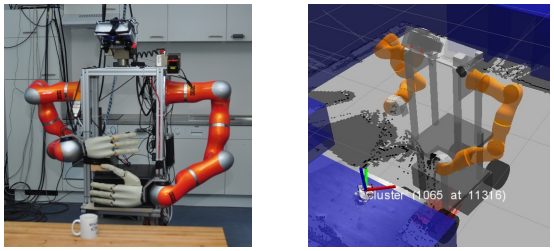

Fig. 3. The robot placed the mug on the table. Thus, it beliefs that the occasion *On(Mug, Table)* holds.

The task tree provides information about the control flow at a high level of detail and contains all stack frames that have been generated from all control structures. This also includes loops, variable assignments and call to helper functions. Plans (i.e. procedures with first-order annotations) on the other hand form a more abstract tree that overlays the task tree and contains the information about the semantics of the control program. Figure 4 shows the plan tree of the goal *Achieve(Loc(Obj, Loc))* that places the object *Obj* at the location *Loc*.

In the plan tree, the fact that *Achieve*(s_2) is a child node of *Achieve*(s_1) indicates that it directly helps to achieve s_1. Let us consider a more specific example. Assume that we want to know why the robot didn't pick up the cup that is standing on the table. The corresponding achieve statement is *Achieve(Loc(Obj, Table))* which has three sub-plans: *Perceive(ObjVisible(Obj))*, *Achieve(ObjInHand(Obj))* and *Achieve(ObjPlacedAt(Obj, Table))*. The status of the task that corresponds to the perceive plan might be *Failed* which indicates that the system was unable to see the object.

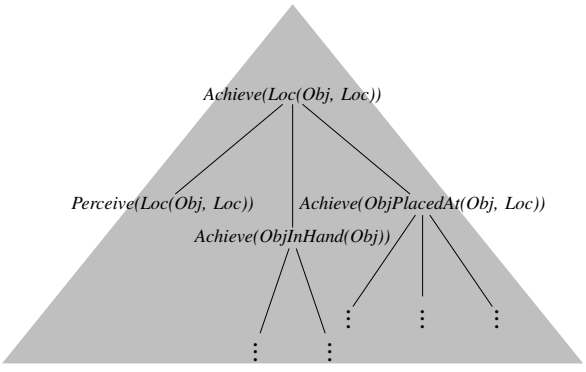

Fig. 4. Example of a plan three that is generated by *Achieve(Loc(Obj, Loc))*. It shows the relationship between the sub-plans. For instance, the reason for executing the statement *Achieve(ObjInHand(Obj))* was to place the object at location *Loc*.

Although being very simple, the example above shows how the concept of a hierarchical plan tree allows for inferring the purpose of a specific plan. Besides the already introduced *Achieve*, plans also contain *Perceive* statements which have a different semantics. Perceiving an occasion means to check if it still holds in the current environment. This does not include any activities that permanently change the environment. For instance, objects are not picked up and placed somewhere else by perceive, but it is possible that a cupboard door is opened to check if an object is still in the cupboard, as long as the door is closed again. Perceive is necessary on control programs that operate robots in dynamic environments that can change without the robot observing the change. That means, a control program needs to deal with wrong belief states, continuously check for consistency and update the belief state when necessary. Thus, every achieve statement normally first perceives if its occasion already holds.

Robot control programs interact with the world based on perception. Therefore, the representation of the real world and, specifically, objects in it are an important property of robot control programs. Furthermore, locations where to perform actions cannot be hard coded, since they depend on the objects that are manipulated, the capabilities of the hardware, for instance the robot's arms, and the environment. The constraints for the motion planner that controls the arms might differ from object to object and must be based on perceived properties such as if the mug is filled or empty.

We represent objects and entities such as parameterizations for the arms and locations as designators, first-order objects that are constructed from a set of symbolic properties describing the entity. For instance, we state the location the robot should stand in order to grasp the mug as

$$graspPos = LocationDesig(Purpose(Grasp), Obj(Mug))$$

instead of a tree dimensional pose. This description is then resolved on demand, not before the real three dimensional coordinates of the location are needed. For example a location to drive to is resolved when the navigation action is performed, not before. The instantiation of a three dimensional pose that satisfies the description can take into account the current context and situation in the belief state. Therefore, it is possible that two designators with the same description resolve to different entities. With designators, we decouple the high-level parameterization of entities describing the interaction in the world from the numeric representation of these parameters. This not only allows for maximal flexibility at execution time but also provides semantic information for reasoning about the control program since the symbolic properties of objects are grounded in the robot's knowledge representation.

As a simple example, let us consider that we want to grasp a white mug that stands on the table. The designators that are involved here are:

1) a location designator that describes the location on the table: *table=LocationDesig(On(Table))*.
2) an object designator that describes the white mug on the table: *mug=ObjectDesig(Type(Mug), Color(White), At(table))*.
3) a trajectory description the arm needs to follow to grasp the mug: *grasp=TrajectoryDesig(To(Grasp), Obj(mug))*

Locations designators are resolved by using a semantic map of the environment. The system uses a connection to the KnowRob [7] knowledge processing system to resolve the symbolic descriptions of the location designator and generate three dimensional poses that can be used, for instance, to navigate to. Object designators are used by the perception subsystem to select search and classification algorithms and build an internal representation of the object. This includes properties that have actually been perceived and the object's pose. Finally, the trajectory designator is translated into a set of constraints that are used by the motion planner to find a motion that fits the description. Please note that this in particular includes context information. For instance, if perception detects that the mug is filled with coffee, it asserts a corresponding property *Contains(Liquid)* to the object designator. This information is used to constrain the trajectory to hold the mug upright.

C. Belief state representation

Let us now consider how the "belief state" of the robot is encoded in control programs. As already mentioned, the belief state is implicitly represented by the variables used in the control program and the execution environment. This includes the set of known objects, including their locations, properties such as color and designators referencing the object, the set of occasions that currently hold and the dynamically evolving task tree including the status of tasks. Dynamic properties such as the current location of the robot within the 2D map of the environment but also the state of tasks are represented in the control program as "Fluents". The plan language we use has special support for fluents. They can be combined to so-called fluent networks, which are still fluents, to represent more complex conditions in the world (see Figure 5).

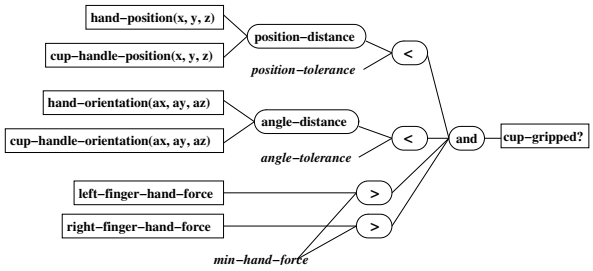

Fig. 5. The fluent network building the fluent *cup-gripped?*.

Fluents are used to wait for conditions and trigger actions. We support this by special expressions in the plan language, namely *WaitFor* to wait for a specific fluent to become true and *Whenever* to execute code whenever a fluent becomes true.

D. System Design

In the previous sections, we have discussed how high level plans are written. In this section we will show how the link to the robot's low level components is made.

The low level components of the robot can be split into the three main modules *perception*, *manipulation* and *navigation*

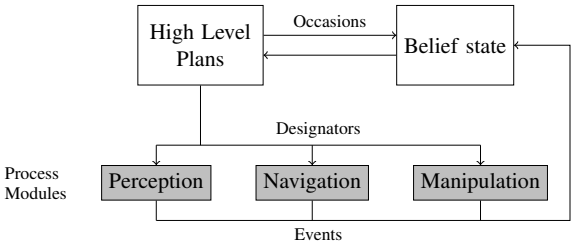

Fig. 6. System Architecture

which are independent of each other. The low level components are implemented as asynchronously running action servers, connected to the system over a middleware, namely ROS [8]. We represent these low level modules in the plan as "Process Modules". A process module provides an interface that is used by high level plans to parameterize it and the low level backend that controls the hardware.

Process modules appear as black boxes to the plan, with a clear and uniform interface. They are parameterized by designators which are resolved inside the process module by taking into account the current situation. Resolving means translating the symbolic description of the designator into numeric control parameters of the underlying system modules that are sent over the middleware. Process modules can be *activated*, *canceled* or transition to the *done* state when their action succeeds. On activation, it reads the designator from the input slot, resolves it and sends the parameters to the low-level process. During execution of the action, the low level process sends feedback information that is accessible to the plan through status and feedback fluents. After successful execution of the action, the result slot is set. Figure 7 shows the navigation process module.

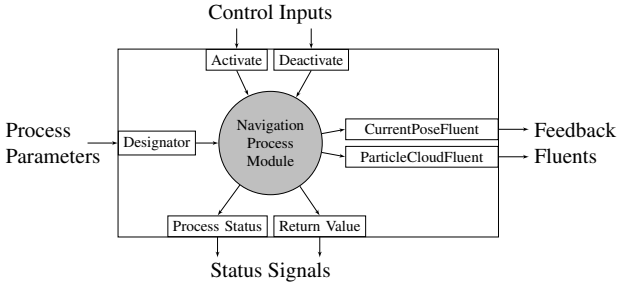

Fig. 7. Process module encapsulating a navigation control process. The input is a location designator, symbolically describing the goal pose and the context this code is executed in (e.g. *LocationDesig(To(see), Obj(Cup))*). The feedback fluents provide information about the status of the current navigation task. The process module can be activated and deactivated and provides success and failure signals.

IV. Recording

Let us continue by taking a look at how logging of the program state works. In CRAM we collect all information relevant to a specific plan execution in a data structure called episode knowledge. This data structure includes the current task tree and an execution trace of changes in the state.

Since all relevant program state is kept in fluents, in order to keep track of state changes we simply log all changes in fluent values at a low level. Whenever a fluents value is changed either by plan execution or by changes from

process modules, CRAM automatically adds a snapshot of the new value together with a timestamp and an unique fluent-identifier to the execution trace. We call this snapshot a traced instance. For example the estimated position of the robot is stored in a fluent as explained above. Whenever the navigation process module updates that fluent, a corresponding traced instance with new position and current timestamp will be logged. Similarly since every task's status is stored in a fluent, each status change will have a corresponding traced instance in the execution trace. In other words the execution trace is the set of all traced instances, which in our implementation is organized in a hash table.

We distinguish between two kinds of episode knowledge which share the same interface for querying the recorded information. Live episode knowledge on the one hand is used during plan execution. It allows for new traced instances to be added to the execution trace. Once plan execution has completed, this live episode knowledge can be saved to disk as offline episode knowledge. The main difference between live and offline episode knowledge is that the former has to account for possible queries while the plan is still executed and thus answers every query by processing the raw traced instances on demand. In contrast the latter pre-processes the logged data and also omits details from the task tree that are needed for execution but not for reasoning[1].

Currently logging is done in memory and persistency is achieved by saving and loading the whole episode knowledge to and from disk. We found that approach to be practical, as neither memory consumption nor performance is an issue at the moment. However the system was designed with extensibility in mind. Possible future extension could be more sophisticated data structures for the execution trace or logging directly to disk[2].

V. Representing Execution Scenarios

Let us now consider the first-order representation of execution scenarios that we use for detailed analysis of the episode. It is based on *occasions* that were to achieve and have been achieved, *events* that are generated from the fluents of the process modules and *intentions* that are encoded in the relationship of plans and their sub-plans within the plan tree and the belief state encoded in the variable values on the execution stack. The concepts of *events* and *occasions* are defined on basic predicates that directly access to the logged data. Table I lists the predicates used in our system.

Asserting Events. Events represent temporal entities of low level control and perception that cause state changes. Most often, events are caused by actions that are performed by the executed plan. We assert the occurrence of an event *ev* at time t_i with *Occurs*(ev, t_i). Events happen at discrete time instances and are generated by the low-level control routines such as the motion planner, arm controllers, tactile sensors in the robot grippers, etc. More specifically, the *Occurs* predicate is calculated from logged fluent values. For instance, when grasping an object, the fluent *cup-gripped?* changes its value and the corresponding event *Collision(Gripper, obj)* is asserted. A successful result of the manipulation process module then leads to the assertion of the event *PickUp(obj)*. Table II summarizes the most important events.

[1]For instance thread resources of the underlying multiprocessing library

[2]This might be useful if memory is scarce on the robots hardware

Occasions and Events	
Holds(occ, t_i)	Occasion assertion in the belief state.
Occurs$(event, t_i)$	Assert the occurrence of an event in the belief state.
Predicates for interfacing the execution log.	
FluentValueAt(fluent, value, t)	Assert the value of a fluent at a specific time.
Task$(task)$	*task* is a task on the interpretation stack.
TaskGoal$(task, goal)$	Unifies the goal of the task.
TaskStart$(task, t)$	Unifies the start time of the task.
TaskEnd$(task, t)$	Unifies the end time of the task.
Subtask$(task, subtask)$	Asserts that *subtask* is a direct subtask of *task*.
Subtask$^+$$(task, subtask)$	Assets that *subtask* is a subtask of *task*.
TaskOutcome$(task, status)$	Unifies the final status of a task (Failed, Done or Evaporated).
TaskResult$(task, result)$	Unifies the result of a task.
TaskFailureDescription$(task, error)$	Unifies the failure object of a failed task.
FailureClass$(error, class)$	Unifies the class of failure.
FailureAttribute$(error, name, value)$	Unifies a slot in a failure object.
DesignatorEqual$(desig_1, desig_2)$	Assert that two designators are equal.
DesignatorValueAt(desig, value, t)	Reads the reference value of a designator at time t.
DesignatorPropertyAt(desig, prop, t)	Asserts a property of the designator at time t.

TABLE I

Basic Predicates.

LocChange(obj)	An object changed its location
LocChange$(Robot)$	The robot changed its location
Collision(obj_1, obj_2)	obj_1 and obj_2 started colliding
CollisionEnd(obj_1, obj_2)	obj_1 and obj_2 stopped colliding
PickUp(obj)	*obj* has been picked up
PutDown(obj)	*obj* has been put down
ObjectPerceived(obj)	The object has been perceived

TABLE II

Event statements.

Asserting States of the World. Occasions are states that hold over time intervals where time instants are intervals without duration. The sentence *Holds*(occ, t_i) represents that the occasion holds at time specification t_i. The term *During*(t_1, t_2) indicates that the occasion holds during any subinterval of the time interval $[t_1, t_2)$ and *Throughout*(t_1, t_2) specifies that the occasion holds throughout the complete time interval. The *Holds* predicate for occasions is defined over events and fluent values. For instance, we define *Holds(ObjectInHand(obj))* by

$$\textit{Holds}(\textit{ObjectInHand}(\textit{Obj}_1), \textit{During}(t_1, t_2)) \Leftrightarrow \exists t_1,\ t_2.\\ \textit{Occurs}(\textit{PickUp}(\textit{Obj}_1), t) \wedge t_1 \leq t < t_2$$

The other occasions are defined accordingly. Table III summarizes the occasions defined in our system. Please note that occasions are not defined on the achieve statements of high level plan structures since these are just annotations of what the specific plan is intended to do. In contrast, we ground occasions as far as possible in the low level data

structures and use the annotation of plans for monitoring and failure handling.

$Loc(obj, loc)$	The location of an object
$Loc(Robot, loc)$	The location of the robot
$ObjectVisible(obj)$	The object is visible to the robot
$ObjectInHand(obj)$	The object is carried by the robot
$ObjectPlacedAt(obj, loc)$	The object has been placed at location.
$TaskStatus(task, status)$	The status of a task.

TABLE III

OCCASION STATEMENTS.

Asserting Intentions. In order to infer the intentions of a plan we have to consider the interpretation stack more carefully. Achieving a state *s* has been an intention if the routine *Achieve(s)* that was on the interpretation stack during the execution. The robot pursued the goal *Achieve(s)* in the interval between the start and the end of the corresponding task. The purpose of achieving *s* can be computed by contemplating the supertasks of *Achieve(s)*. Thus, if we want to answer if there has been a task that navigated the robot in order to grasp an object, we state:

$$\begin{aligned} &Task(tsk) \ \wedge \ TaskGoal(tsk, Achieve(Loc(Robot, loc))) \\ &\wedge \ Task(super) \ \wedge \ Subtask^{+}(super, task) \\ &\wedge \ TaskGoal(super, Achieve(ObjectInHand(obj))) \end{aligned}$$

VI. EVALUATION

In this paper we have explained a reasoning mechanism that equips a robot control program with additional functionality and information that can be used to explain past action scenarios but that is also available at runtime. The power of our approach lies in the number of different questions that can be answered, i.e. the number of Prolog clauses that can be defined over the predicates introduced above. In the following, we illustrate the expressiveness by couple of examples.

As a basic query, we might ask the robot if it grasped any green object, by this statement:

$$\begin{aligned} &Task(tsk) \ \wedge \ TaskGoal(tsk, Achieve(ObjectInHand(obj))) \\ &\wedge \ TaskEnd(tsk, t) \\ &\wedge \ DesignatorPropertyAt(obj, Color(Green), t) \end{aligned}$$

Please note that this does not imply that it was specified in the plan that the robot should grasp a green object. The reason is that before grasping it, the robot searches for the object and asserts additional properties if the perception routines provide the corresponding information. That explains also why we need a time parameter in the predicate *DesignatorPropertyAt* since the description matching a specific object may change during the course of action. For instance, a cup can be filled at the beginning but empty later on.

Other queries we can answer include whether the robot manipulated the same object twice. For instance, if the robot is to achieve that one cup is on the table and another cup is on the counter, we can detect the failure that it first placed a cup on the table and then put the same cup on the counter. Obviously, this is unwanted behavior that cannot be detected by program exceptions in a general way. With our first-order representation, this can be expressed by asserting an error if an occasion that has been achieved by a top level task does not hold at the end:

$$\begin{aligned} &UnachievedGoalError(o) \ \Leftrightarrow \\ &\quad TopLevel(tt) \ \wedge \ Task(tsk) \wedge \ Subtask(tt, tsk) \\ &\quad \wedge \ TaskGoal(tsk, o) \ \wedge \ TaskEnd(tt, t) \\ &\quad \wedge \ \neg Holds(o, t) \end{aligned}$$

The predicate *TopLevel* is defined to unify the task that is not the subtask of any other task. Please not that we use *SubTask* instead of $SubTask^{+}$ to denote only direct subtasks of the toplevel. Recording the execution log has no negative influence on the performance of plan execution. That means although all information that is necessary to completely reconstruct plan execution is saved, execution time is not increased noticeable. Answering complex queries that involve data structures that change frequently, such as the location of the robot, take less than 5 seconds. By careful implementation of the predicates it is even possible to do light weight inferences within fast control loops. More time consuming inferences can be made at runtime without loss of time if they are parallelized, for instance if they are executed while the robot is navigating to a location.

VII. CONCLUSIONS

In this paper, we presented an extension to CRAM that implements high performance and accurate reasoning mechanisms. These allow to answer complex questions about plan execution, the intention of the robot, the reason for failures and the belief state of the robot. This is achieved by creating an extensive execution trace and mechanisms to query it through a first-order representation. Since this system can not only be used offline but also during plan execution, i.e. within control routines, it enables deep and complex failure handling mechanisms that are based on descriptions of failures in the first-order representation.

Acknowledgements: The research reported in this paper is supported by the cluster of excellence COTESYS (Cognition for Technical Systems, *www.cotesys.org*).

REFERENCES

[1] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments," in *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2010, accepted for publication.

[2] R. Simmons, D. Goldberg, A. Goode, M. Montemerlo, N. Roy, B. Sellner, C. Urmson, M. Bugajska, M. Coblenz, M. Macmahon, D. Perzanowski, I. Horswill, R. Zubek, D. Kortenkamp, B. Wolfe, T. Milam, and B. Maxwell, "Grace: An autonomous robot for the aaai robot challenge," 2002.

[3] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994.

[4] M. Beetz, "Structured Reactive Controllers," *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Best Papers of the International Conference on Autonomous Agents '99*, vol. 4, pp. 25–55, March/June 2001.

[5] M. Beetz and D. McDermott, "Declarative goals in reactive plans," in *First International Conference on AI Planning Systems*, J. Hendler, Ed., Morgan Kaufmann, 1992, pp. 3–12.

[6] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnstrom, "Temporal action logics (tal): Language specification and tutorial," 1998.

[7] M. Tenorth and M. Beetz, "KnowRob — Knowledge Processing for Autonomous Personal Robots," in *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2009.

[8] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *In IEEE International Conference on Robotics and Automation (ICRA 2009)*, 2009.