ROSoClingo: A ROS package for ASP-based robot control

Benjamin Andres, Philipp Obermeier, Orkunt Sabuncu and Torsten Schaub
Institute for Computer Science,
University of Potsdam, Germany
Email: {bandres,phil,orkunt,torsten}@cs.uni-potsdam.de
David Rajaratnam
School of Computer Science and Engineering
The University of New South Wales, Australia
Email: daver@cse.unsw.edu.au

***Abstract*—Knowledge representation and reasoning capacities are vital to cognitive robotics because they provide higher level cognitive functions for reasoning about actions, environments, goals, perception, etc. Although Answer Set Programming (ASP) is well suited for modelling such functions, there was so far no seamless way to use ASP in a robotic environment. We address this shortcoming and show how a recently developed reactive ASP system can be harnessed to provide appropriate reasoning capacities within a robotic system. To be more precise, we furnish a package integrating the reactive ASP solver *oClingo* with the popular open-source robotic middleware ROS. The resulting system, *ROSoClingo*, provides a generic way by which an ASP program can be used to control the behaviour of a robot and to respond to the results of the robot's actions.**

I. Introduction

Knowledge representation and reasoning capacities are vital to cognitive robotics because they provide higher level cognitive functions for reasoning about actions, environments, goals, perception, etc. Although Answer Set Programming (ASP) is well suited for modelling such functions, there was so far no seamless way to use ASP in a robotic environment. This is because ASP solvers were designed as one-shot problem solvers and thus lacked any reactive capacities. So, for instance, each time new information arrived, the solving process had to be re-started from scratch.

In what follows, we address this shortcoming and show how a recently developed *reactive* ASP system [7], [6] can be harnessed to provide knowledge representation and reasoning capacities within a robotic system. This is possible because such systems allow for incorporating online information into operative ASP solving processes. We accomplish this by integrating our ASP approach into the popular open-source middleware ROS (Robot Operating System; [13][1]) which has become a de facto standard in robotics over the last years. As such, ROS provides hardware abstraction and tools supporting the development of robot applications.

To be more precise, we furnish a ROS package integrating the reactive ASP solver *oClingo* with the popular open-source ROS robotic middleware. The resulting system, called *ROSoClingo*, provides a generic method by which an ASP program can be used to control the behaviour of a robot and to respond to the results of the robot's actions. In this way, the *ROSoClingo* package plays the central role in fulfilling the need for high-level knowledge representation and reasoning in cognitive robotics by making details of integrating a highly capable reasoning framework within a ROS based system transparent for developers. In what follows, we provide the architecture and basic functioning of the *ROSoClingo* system. And we illustrate its operation via a case-study conducted with the ROS-based *TurtleBot*[2] simulation in a *Gazebo*[3] simulation of an office floor.

The Golog programming language [11] is one of the most widely known approaches to the development of a declarative agent reasoning language. With a formal semantics based on the Situation Calculus [12] it allows for the specification of high-level agent behaviours for agents acting within dynamically changing environments. The potential power of this approach was first shown on a real robot with the implementation of the Golex system [10]. Golex extended Golog with execution monitoring functionality to monitor and ensure the successful execution of the primitive Golog actions.

With the success of Golog, further work has focused on extending ASP with some of the expressive constructs found in Golog [14], thus allowing the powerful search capabilities of modern reasoners to be combined with the programming ease of Golog. The development of *ROSoClingo* can therefore be understood in the context of allowing Golog, and other, ASP extensions for agent reasoning to be directly applied to the high-level control of ROS based robots.

Finally, we list some related work which utilize ASP or other declarative formalisms in cognitive robotics. The work in [3], [2] uses ASP for representing knowledge via a natural language based human robot interface. Additionally, ASP is used for high level task planning. In [1], [5] action language formalism and ASP are used to plan and coordinate multiple robots for fulfilling an overall task. They have also integrated

[0]This paper is also submitted at the Knowledge Representation and Reasoning in Robotics Workshop at ICLP 2013

[1]`http://www.ros.org`

[2]`http://turtlebot.com`

[3]`http://gazebosim.org`

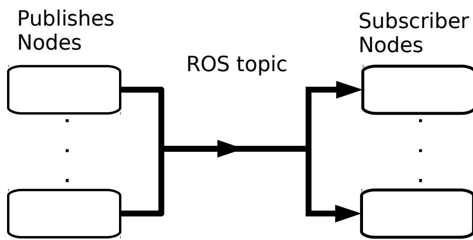

Fig. 1. ROS topics provide many-to-many asynchronous message passing.

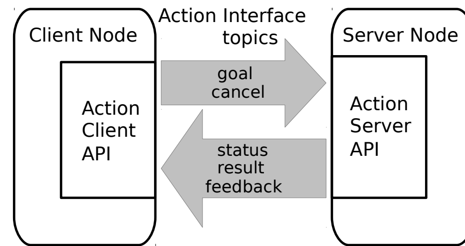

Fig. 2. ROS actions provide preemptible client-server communications.

task and motion planning with external calls from action formalism to geometric reasoning modules [4]. All these works can naturally and highly benefit from the usage of *ROSoClingo*. Having stated that, *ROSoClingo* can be basically used in any autonomous robotics system in which high-level reasoning tasks are essential and steep initial integration difficulties are desired to be avoided.

II. Robot Operating System

ROS provides a middleware for robotic applications [13]. At its most basic level this consists of a loosely-coupled communication framework for sending *messages* between processes. ROS defines a host and language independent TCP/IP protocol for exchanging messages, thus allowing these processes to be written in a variety of programming languages and to be distributed across multiple host computers.

ROS standardises methods and structures for organising software into *packages*. Packages can contain a variety of components, from definitions of message formats through to libraries and executable programs. Executable programs that integrate into the ROS framework are instantiated as special processes known as *nodes*.

There are two basic mechanisms for communications between nodes. The *publisher-subscriber* mechanism provides for asynchronous communications whereby multiple nodes can broadcast messages on a named communication channel (known as a *topic*), that are in-turn listened to by multiple subscribers (Figure 1). Alternatively, ROS *services* provide for synchronous communication via a remote procedure call (RPC) mechanism whereby one node can call a service provided by another node.

All messages in ROS are strongly-typed, and all communications using topics or services must use these types. ROS provides a number of primitive data types (e.g., `bool`, `int32`, `float32`, `string`) as well as a list operator. These can be combined to produce arbitrarily complex types in a similar manner to `structs` in C and C++. These complex data types are defined as part of the ROS package structure. To ease development and code maintenance ROS package names inherently correspond to namespaces of the same name and therefore the complex data types are always defined with respect to a namespace. A typical ROS system defines a number of common namespaces (e.g., `std_msgs`, `geometry_msgs`) and data types (e.g., `geometry_msgs/Pose`).

While ROS services allow for a simple RPC mechanism, they are not suitable for more complex behaviours, such as situations where a task may take place over an extended time frame, may be preempted, and may require feedback throughout its life-cycle. ROS provides for such complex behaviour through the *actionlib* framework (Figure 2). This framework allows a client-server interface to be defined whereby an *action client* is able to set and cancel goals on an *action server*. The action server in turn executes the goal and provides constant feedback and progress of its attempts to fulfil the goal. While implemented using ROS topics as a message transport mechanism, each *action interface* defines a high-level API for client-server interaction. [4]

As a prototypical example of a ROS action, the `move_base` package implements an action interface that provides path-planning and robot control functionality for moving a robot around an environment. We now briefly outline this package, to provide both a sense of how the *actionlib* framework works as well as to provide details of an important module that we shall discuss later.

The `move_base` package (Figure 3) provides a highly configurable ROS node that is an essential component of the ROS *navigation stack*. The system requires a number of data sources, such as laser sensor data, localisation information, and map information. The laser sensor data is used to perform basic obstacle avoidance, while localisation information allows the robot to know where it is located within the map. A map consists of a 2-D *occupancy grid* that indicates whether a point on the grid is occupied or free. Robot navigation within the node takes place at two distinct levels. *Global planning* calculates the route from the robot's current location to a goal location, while *local planning* provides for movement towards a general direction while allowing path flexibility to avoid obstacles. A range of different global and local planning algorithms are supported through a plugin architecture.

A navigation goal is specified in terms of a robot destination *pose* (i.e., position and orientation). When a goal is sent to the `move_base` server it computes a path to that goal location and then successively generates the movement commands for the robot base controller. If at some point the robot is unable to proceed with its plan, for example due to a door being blocked, then the server will undertake recovery behaviour and will re-plan accordingly. If the recovery fails then the task will be aborted. Throughout this process the server provides constant feedback as to the current location of the robot, as well as the

[4]For more extensive information on programming with the ROS *actionlib* framework the interested reader is referred to `http://www.ros.org/wiki/actionlib`

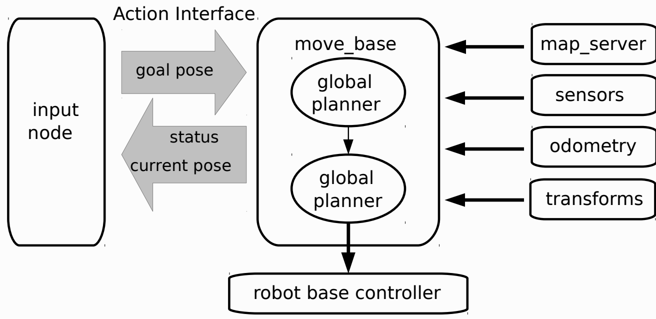

Fig. 3. ROS `move_base` provide an action interface for robot movement.

status of the navigation process, for example that the task has been aborted. Goals are preemptible, so that a robot navigating towards some location will give up that goal if it is given a new destination goal.

The action interface provided by the `move_base` package is arguably the most important ROS action service for mobile robots. Furthermore, with a complex set of features, such as the possibility of failure, it serves to highlight the potential complexity in trying to integrate logical reasoning with a real robotic systems. Consequently, the `move_base` package forms much of the integration work that is outlined in the rest of this paper.

A. oClingo

A classical ASP system, such as gringo/clasp [9], [8], is designed to solve problems in a one-shot procedure: it takes a problem encoding as input, computes the answer sets and terminates afterwards. Since this approach does not fully embrace the needs of modern dynamic domains, such as robotics, a reactive ASP solver, *oClingo* [7], [6], was developed that additionally takes external data streams into account. Such a stream is represented there by an *online progression*, a sequence of events and inquiries given in the form of ASP ground facts and integrity constraints. The general problem itself is described by a *reactive logic program*, an ASP program that is partitioned into three parts: a *base* part describing static knowledge, and an *incremental* as well as a *volatile* part which both contain rule schemata based on a discrete time (integer) parameter. The role of the incremental part is to symbolize accumulated knowledge over increasing time, whereas the volatile part only holds information that specifically concerns the current point in time. All in all, a reactive logic program formulates the persistent knowledge and, thus, acts as the offline counterpart to an online progression.

Technically, the *oClingo* system is initialized with a reactive logic program as input. Afterwards, an application can connect to *oClingo* and send a data stream, formatted as an online progression. For each incoming stream update *oClingo* computes all answers, returns them to the client and subsequently waits for the next input. The *oClingo* system only terminates if explicitly requested by the user.

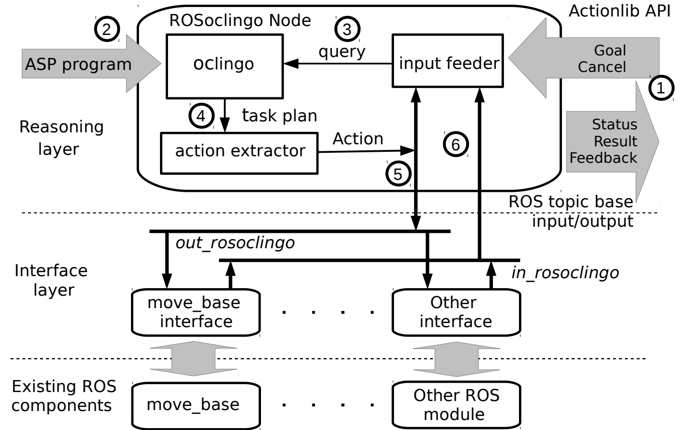

Fig. 4. The general architecture and main workflow of *ROSoClingo*.

III. *ROSoClingo*

In this section we describe the general architecture and functionality of the *ROSoClingo* package. With the help of reactive ASP, *ROSoClingo* provides a way of handling high-level knowledge representation and reasoning tasks occurring in autonomous robots running the ROS software.

Consider a task planning problem, a task that any autonomous robot should be capable of performing. For instance, a robot is given a goal of moving from the kitchen of a house to the living room in order to serve food. Even if the robot has many individual behaviours, like moving from one point to another or holding food with the help of respective ROS packages, computing a complete task plan requires high-level knowledge representation and reasoning capabilities. There might even be more than one possible path to the living room, which may require more elaborate planning and execution. The robot might, for instance, need to first go to the hallway and then to the living room in case the alternative path is blocked by an obstacle. The robotics developer can encode such planning tasks in reactive ASP keeping only the interface requirements of the underlying behaviour nodes in mind and avoiding implementation details of their functionality (motion planning for example). Then, using the resulting ASP encoding with the *ROSoClingo* package the developer can readily integrate task planning while details of controlling and integrating *oClingo* within the ROS middleware becomes transparent.

Figure 4 depicts the main components and workflow of the *ROSoClingo* system. It consists of a three layered architecture. The first layer consists of the core *ROSoClingo* component and the definition of an *actionlib* API. This API allows other components to use the services provided by the *ROSoClingo* node. The package also defines the message structure for communication between the core *ROSoClingo* node and the various nodes of the interface layer. The interface layer, on the other hand, provides the data translations between what is required by the *ROSoClingo* node and any ROS components for which it needs to integrate. This architecture provides for a clean separation of duties, with the well-defined abstract reasoning tasks handled by the core node and the integration details handled by the interface nodes.

`_request(Goal,C)`	Specifying a Goal request at cycle `C`.
	Cancellations are send in the same way.
`_action_lib(A,P,C)`	Commanding *actionlib* `A` to run with parameters `P` at cycle `C`.
`_return(A,V,C)`	Specifying the return value `V` of *actionlib* `A` run in cycle `C`.

Fig. 5. Keywords used for communicating between *ROSoClingo*and *oClingo*.

A. The ROSoClingo Core

The main *ROSoClingo* node is composed of the reactive answer set solver *oClingo*, an action extractor, and an input feeder. Through its *actionlib* API, it can receive goal and cancellation requests as well as send result, feedback, and status messages to a client node (marked by 1 in the figure). The reactive ASP program, encoding the high-level task planning problem, is given to the *ROSoClingo* node at system initialization (marked by 2). During initialization, *ROSoClingo* sets the current logical time point to 1. This time point is incremented at the end of each cycle.

A cycle of *ROSoClingo*'s workflow may start with a goal for the robot arriving via the *actionlib* interface (marked by 1). For instance, commanding the robot to go to the living room can be a goal request. This request is transformed into an input stream update before feeding to *oClingo* (marked by 3) by the input feeder. *oClingo* receives the goal as a stream update and searches for an answer set representing a task plan for the robot to follow. Each action of the plan, in principle, should be executed by a respective ROS node. For instance, an action of moving to the door connecting kitchen and hallway can be executed by the `move_base` ROS action node. The keywords of table 5 allow for a communication protocol between *ROSoClingo*and *oClingo*. The action extractor takes the action at the current logical time point, prepares it as a goal request (marked by 4) and sends it to be executed by the respective ROS node (marked by 5). The result of the execution is received by the input feeder component of the *ROSoClingo* node (marked by 6). The communication between *ROSoClingo* and other ROS nodes is detailed in Section III-B. The result is processed and transformed into a new input input stream update for *oClingo*, which completes the current and initiates a new cycle of *ROSoClingo*. The (un)successful result may generate new knowledge for the robot about the world (for example, the fact that a doorway is blocked or a new object is sensed). Additionally, the next input update includes a fact so that the action executed is committed by *oClingo* during further searches for a plan.

Note that the *ROSoClingo* package supports multiple goal requests at a time. Each time a new goal received, the input feeder appends the goal to a list and feeds it to *oClingo* in the next cycle. The status of each goal is also tracked by *ROSoClingo*.

B. Integrating with Existing ROS Components

The core *ROSoClingo* node needs to issue commands to, and receive feedback from, existing robotics components. The complexity of this interaction is handled by the nodes at the interface layer (Figure 4). Unlike the components of the reasoning layer it is, unfortunately, not possible to define a single ROS interface to capture all interactions that may need to take place. Firstly, there will need to be data type conversions between the individual modules. For example, the `move_base` node expects a robot *pose* as its goal, while an action to move a robot arm might require more a complex goal structure consisting of a set of joint-trajectories. Turning ROS messages into a suitable set of *oClingo* statements will therefore require data type conversions that are specific for each action or service type.

A second complicating issue is that the level of abstraction of a ROS action may not be at the appropriate level required by the ASP program. For example, the *pose* goal for moving a robot consists of a Cartesian coordinate and orientation. However, it is unlikely that one would want a logical reasoner to have to reason about Cartesian coordinates. Instead one would hope to reason about abstract locations and the relationship between these locations; for example that the robot should navigate from the kitchen to the bedroom via the hallway. Furthermore, the desired orientation of the robot when it arrives in the bedroom may not be something that is of interest to the reasoner.

While it is not possible to provide a single generic interface to all ROS components, it is however possible to outline a common pattern for such integration. The rest of this section outlines the integration of *ROSoClingo* with ROS actions, and in particular the `move_base` action outlined in Section II. ROS actions typically encapsulate the high-level behaviour and functionality of a robot, and are therefore the most natural level at which a high-level robot controller would expect to communicate with the rest of the robotic system. Furthermore, they are arguably the most complex components of a ROS system. Consequently, showing how *ROSoClingo* integrates with existing ROS actions encapsulates all the complexity that one would expect of integration with any other ROS component.

For each existing ROS component that needs to be integrated with *ROSoClingo* there will need to be a corresponding interface component. In some cases interface components can be combined into a single ROS node to communicate with multiple lower-level ROS nodes, but in general one can imagine a mostly one-to-one correspondence between nodes of these two layers.

An important consequence of our architecture is that every interface node needs to read every message that is published by the *ROSoClingo* node on the output topic. It is therefore important that the message format for the *out_rosoclingo* topic allows the interface components to easily parse the messages and discard those messages that are intended for a different component.

The inputs to, and outputs from, a running *oClingo* reasoner consist of sets of facts. It is therefore the role of

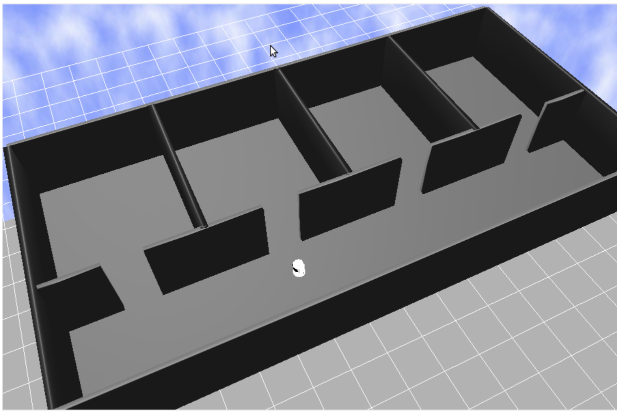

Fig. 6. The mailbot simulation in progress as seen in *Gazebo*.

the ROS interface layer to perform any data conversions between the ASP world of facts and the low-level ROS commands. We adopt a straightforward message type (named `rosoclingo/InterfaceIO`) to facilitate this process. This type consists of an interface name and a list of text formatted facts.

When a message is sent from *ROSoClingo* to the interface layer, individual interface components can quickly parse the interface name to determine the intended recipient and discard non-relevant messages. On the other hand, when sent from an interface component to the *ROSoClingo* node, the interface name indicates the origin of the fact, which may be useful, even if only as a debugging aid.

For the sake of simplicity and presentation it is useful to make some assumptions, in showing how the system integrates with the `move_base` action. A common assumption in robotic applications is to identify tagged points with an abstract location. For example some coordinate location specifying a point in a bedroom, say `(10.5,11.2)`, will be associated with the label "bedroom". Navigation can then take place with reference to the tagged locations. This technique works for a broad range of behaviours, such as sending the robot to specific locations.

IV. Case Study

We demonstrate the application of our *ROSoClingo* package on a mail delivery scenario running ROS software. The scenario consists of a robot, whose task is to pick up and deliver mail packages exchanged among offices [15]. Whenever a mail delivery request is received, the robot has to go to the office requesting the delivery, pick up the mail package, and go to the destination office in order to do the delivery. In this scenario the robot is able to carry up to three packages and handle multiple requests at a time. Delivery requests can also be cancelled during task execution. If a request is cancelled when the package has already been picked up, it is delivered back to its origin for disposal. The task has a highly dynamic nature and requires reasoning capacity for detailed planning.

The robot we use is a *TurtleBot*, which is well supported within the ROS community and commonly used for small delivery tasks. Offering a mobile platform with an integrated Microsoft Kinect as a three dimensional sensor. Our office building is provided by *Gazebo*, a simulator supported by ROS, able to realistically simulate three dimensional environments. This allows us to run the *TurtleBot* in a controlled, yet physically plausible environment, while avoiding all too common problems associated with hardware, e.g. short battery life, defunct components, etc.

We use the ROS `move_base` action library for robot movement among offices. For picking up and delivering packages, two dedicated action libraries are used (`pickup` and `deliver`). Figure 6 shows the *Gazebo* environment used in our case study. It consists of 4 consecutive offices on one floor; `office1` to `office4` appearing from left to right.

In our scenario the robot stands in front of `office1` and after some time receives a request to deliver a package from `office3` to `office2`. This request is later cancelled and a new request to deliver a package from `office3` to `office4` is issued.

Since there are no pending requests just after the start of the simulation, *oClingo* returns an empty task plan. This results in *ROSoClingo* awaiting a new request to be issued. When the first request is received, the input feeder transforms said request into an input stream update for *oClingo* :

```
#step 1.
_request(goal(office3,office2,1),2).
#end step.
```

With "`#step 1.`" identifying the start of an input stream update and the cycle the request is send to *oClingo*. The keyword predicate "`_request`" identifies the transformed goal request issued to *ROSoClingo* with its first parameter and the cycle the request becomes active with its second. The parameters of "`goal`" state the sending office, the destination office and an unique package identifier, in that order. "`#end step.`" closes the input stream update.

oClingo now adapts the task plan to ensure the execution of the request as shown in Table I under plan 1. In more detail the plan involves to use the `move_base` action library to move the robot from `office1` over `office2` to `office3` at the cycles 1 and 2, respectively. Then, the robot shall use the `pickup` action library to pick up the package in cycle 3 and move back to `office2` in cycle 4. Lastly, the package is to be handed over by means of the `deliver` action library in the 5. cycle. Note, that both the `pickup` and the `deliver` action require the package identifier as parameter.

ROSoClingo's action extractor takes the task plan from *oClingo* and publishes the action planned for the current (first) cycle on the *out_rosoclingo* topic as a `rosoclingo/InterfaceIO` message. The `move_base` interface reacts to the message, transforms the label `office2` into a coordinate location and sends the result of the action back to *ROSoClingo* via the *in_rosoclingo* topic.

In the next cycle the result is feed back into *oClingo* using the keyword predicate "`_return`". Assuming `move_base` was successful, the input feeder generates the following input stream update for the second cycle:

TABLE I
TASK PLANS RETURNED BY *oClingo* FOR VARYING REQUESTS.

plan	1	2	3
step			
1	`_action_lib(move_base,office2,1)`	`_action_lib(move_base,office2,1)`	`_action_lib(move_base,office2,1)`
2	`_action_lib(move_base,office3,2)`	`_action_lib(move_base,office3,2)`	`_action_lib(move_base,office3,2)`
3	`_action_lib(pickup,1,3)`		
4	`_action_lib(move_base,office2,4)`		`_action_lib(pickup,2,4)`
5	`_action_lib(deliver,1,5)`		`_action_lib(move_base,office4,5)`
6			`_action_lib(deliver,2,6)`

```
#step 2.
:- not _action_lib(move_base,office2,1).
_return(move_base,office2,1).
#end step.
```

The integrity constraint after the input stream header enforces *oClingo* to include the action just taken into future action plans. Otherwise, *oClingo* might abolish actions taken in the past in order to minimize the task plan. The rest of the cycle runs analogous to the first cycle shown. At sometime between the 2 and 3 cycle *ROSoClingo* receives the cancellation of the first request:

```
#step 3.
:- not _action_lib(move_base,office3,2).
_request(cancel(1),3).
_return(move_base,office3,2).
#end step.
```

With "`cancel(1)`" identifying the delivery to be cancelled via its package identifier. This forces *oClingo* to change the task plan to the one presented in Table I under plan 2. Since now there are no actions planned for the current (third) cycle the robot waits idly at `office3` for new requests. When *ROSoClingo* receives the second request, the input feeder generates a new input stream update for *oClingo* initiating the fourth cycle. The task plan generated by *oClingo* for satisfying the request is shown in Table I under plan 3. Again, assuming the actions are executed without complications the following cycles run analogous to the ones above. After the sixth cycle the robot delivered the package to `office4` and enters the idle mode again, awaiting new requests.

V. CONCLUSION

Higher level cognitive functions such as reasoning about actions, environment, goals, or perceptions are crucial in cognitive robotics. They necessitate knowledge representation and reasoning capacities for autonomous robots. We developed a ROS package integrating *oClingo*, a reactive ASP solver, with the robotics middleware ROS. The resulting system, called *ROSoClingo*, fulfils the need for high-level knowledge representation and reasoning in cognitive robotics by providing a highly expressive and capable reasoning framework. It also makes details of integrating *oClingo* transparent for the developer. Using reactive ASP and *ROSoClingo*, one can control the behaviour of a robot within one framework and in a fully declarative way. This is particularly important compared to Golog based approaches where the developer should take care of implementation (usually in Prolog) details of the control knowledge and the underlying action formalism separately. We illustrated the usage of *ROSoClingo* via a case-study conducted with a ROS-based simulation of a robot delivering mail packages in an office environment using *Gazebo*.

The resulting work is publicly available and we are committed to submit the *ROSoClingo* package to the public ROS repository.

Acknowledgments. This research was partly supported under ARC Discovery Projects funding scheme (project number DP 120102144) and the DFG grant SCHA 550/9-1.

REFERENCES

[1] E. Aker, A. Erdogan, E. Erdem, and V. Patoglu. Causal reasoning for planning and coordination of multiple housekeeping robots. pages 311–316.

[2] X. Chen, J. Ji, J. Jiang, G. Jin, F. Wang, and J. Xie. Developing high-level cognitive functions for service robots. pages 989–996.

[3] X. Chen, J. Jiang, J. Ji, G. Jin, and F. Wang. Integrating NLP with reasoning about actions for autonomous agents communicating with humans. pages 137–140.

[4] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. pages 4575–4581.

[5] Esra Erdem, Erdi Aker, and Volkan Patoglu. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, 5(4):275–291, 2012.

[6] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Stream reasoning with answer set programming: Preliminary report. pages 613–617.

[7] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. Reactive answer set programming. pages 54–66.

[8] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. pages 260–265.

[9] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. pages 266–271.

[10] Dirk Hähnel, Wolfram Burgard, and Gerhard Lakemeyer. GOLEX - bridging the gap between logic (GOLOG) and a real robot. In Otthein Herzog and Andreas Günter, editors, *KI*, volume 1504 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 1998.

[11] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[12] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[13] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler A., and Ng. ROS: an open-source robot operating system. In *ICRA Workshop on OSS*, 2009.

[14] Tran Cao Son, Chitta Baral, and Sheila A. McIlraith. Extending answer set planning with sequence, conditional, loop, non-deterministic choice, and procedure constructs. In Alessandro Provetti and Tran Cao Son, editors, *Answer Set Programming*, 2001.

[15] Michael Thielscher. Logic-based agents and the frame problem: A case for progression. In V. Hendricks, editor, *First-Order Logic Revisited: Proceedings of the Conference 75 Years of First Order Logic (FOL75)*, pages 323–336, 2004.