Integrating ASP into ROS for Reasoning in Robots

Benjamin Andres[1], David Rajaratnam[2], Orkunt Sabuncu[1], and Torsten Schaub[1,3⋆]

[1] University of Potsdam
[2] University of New South Wales
[3] INRIA Rennes

Abstract. Knowledge representation and reasoning capacities are vital to cognitive robotics because they provide higher level functionalities 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 setting. We address this shortcoming and show how a recently developed ASP system can be harnessed to provide appropriate reasoning capacities within a robotic system. To be more precise, we furnish a package integrating the new version of the ASP solver *clingo* with the popular open-source robotic middleware Robot Operating System (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.

1 Introduction

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

In this paper, we address such shortcomings and show how a recently developed (multi-shot) ASP system [1] can be harnessed to provide knowledge representation and reasoning capabilities within a robotic system. We accomplish this by integrating a multi-shot ASP approach, where online information can be incorporated into an operative ASP solving process, into the popular open-source middleware ROS[4] (Robot Operating System; [2]).

To be more precise, we furnish a ROS package integrating the ASP solver *clingo* 4 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 of fulfilling the need for high-level knowledge representation and reasoning in cognitive robotics by making details of integrating a reasoning framework within a ROS based system transparent to developers.

⋆ Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.
[4] `http://www.ros.org`

As we detail below, the robotics developer can encode high-level planning tasks in ASP keeping only the interface requirements of the underlying behaviour nodes in mind and avoiding implementation details of their functionality (motion planning for example).

One crucial added value of our integration of reactive ASP framework into ROS is the facility of encoding adaptive behaviours directly in a declarative knowledge representation formalism. Additionally, the robot programmer can handle execution failures directly in the reasoning formalism. This paves the way for deducing new knowledge about the environment or diagnostic reasoning in the light of execution failures. The case study in Section 4 demonstrates these advantages of *ROSoClingo*.

Finally, it is worth mentioning a number of related approaches which utilize ASP or other declarative formalisms in cognitive robotics. In the work of [3, 4] ASP is used for representing knowledge via a natural language based human robot interface. Additionally, action language formalisms and ASP have been used to plan and coordinate multiple robots for fulfilling an overall task [5, 6]. ASP has also been used to integrate task and motion planning via external calls from action formalism to geometric reasoning modules [7]. However, all these implementations rely on one-shot ASP solvers and thus lack any reactive capabilities. Hence, they could greatly benefit from the reactive solving that comes from the usage of *ROSoClingo*.

In what follows, we provide the architecture and basic functionality of the *ROSoClingo* system. We then outline the ASP encoding for an example mail delivery robot. This example serves to highlight the features of the system but also serves as a guide for how an ASP encoding could be written for other application domains. The operations of the mail delivery robot are illustrated via a case-study conducted within a 3D simulation environment.[5] The features of *ROSoClingo* are discussed with reference to this case study and through comparisons to alternative approaches. Finally, it should be mentioned that the *ROSoClingo* system is publicly available [13] and we are committed to submitting the *ROSoClingo* package to the public ROS repository.

2 *ROSoClingo*

In this section, we describe the general architecture and functionality of the *ROSoClingo* system. With the help of the reactive ASP solver *clingo* (version 4), *ROSoClingo* provides high-level knowledge representation and reasoning capabilities to ROS based autonomous robots. Critically, *clingo* supports multi-shot reactive solving, where the solver does not simply terminate after an initial answer set computation, but instead enters a loop, incrementally incorporating new information into the solving process. For more extensive background to both *clingo* 4 and ROS the interested reader is referred to an extended version of this paper [8].

Figure 1 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 instantiation of a ROS *actionlib* API. In essence, this API simply exposes the services provided by *ROSoClingo* for use by other processes (i.e., ROS *nodes*) . The package also defines the message structure for communication between

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

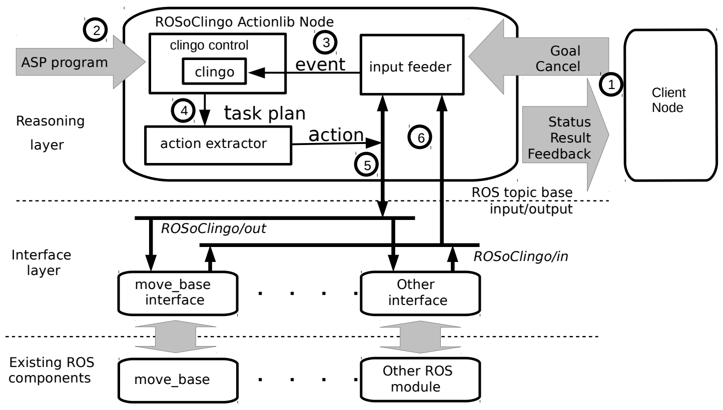

Fig. 1. The general architecture and main work flow of *ROSoClingo*.

the core *ROSoClingo* node and the various nodes of the interface layer. In contrast to the reasoning layer, the interface layer 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.

2.1 The *ROSoClingo* Core

The main *ROSoClingo* node is composed of a python module for the answer set solver *clingo* controlled by `clingoControl`, an `actionExtractor`, and an `inputFeeder`. Through its ROS *actionlib* API, it can receive goal and cancellation requests as well as send result, feedback, and status information back to a client node (marked by 1 in Figure 1). The ASP program, encoding the high-level task planning problem, is given to the *ROSoClingo* node at system initialization (marked by 2). During initialization, *ROSoClingo* grounds the base subprogram of the ASP encoding and sets the current logical time point as well as the current horizon to 0. The logical time point identifies which actions of a task plan are to be executed next, while the horizon identifies the length of the task plan. The time point is incremented at the end of each cycle.

ROSoClingo's workflow starts with a goal arriving at the `inputFeeder` (marked by 1). If *clingo* is already in the process of searching for a task plan, the solving procedure is interrupted and the new goal is added to the solver. The goal request is transformed into an ASP fact and transmitted to *clingo* (marked by 3). Then `clingoControl` is called to resume the solving process with the additional goal.

Algorithm 1 presents the pseudo code representation of the `clingoControl` procedure. The *clingo* functions `assignExternal` as well as `ground` are explained in more detail in Section 3. It instructs the *clingo* solver to asynchronously find a task plan that satisfies all given goals. If *clingo* is able to find a valid task plan then the solution is forwarded to the `actionExtractor`. If no task plan is found for the current horizon,

Algorithm 1: `clingoControl`

```
solveAsynchronous
if clingo returns satisfiable then
    task plan ← get answer set from clingo
    actionExtractor(task plan)
if clingo returns unsatisfiable then
    horizon ← horizon + 1
    assignExternal(Fun("horizon",[horizon-1]),False)
    ground([("transition",[horizon])])
    ground([("query",[horizon])])
    assignExternal(Fun("horizon",[horizon]),True)
    clingoControl
```

`occurs(Robot,Action,T)`	Out	Commanding the robot `Robot` to execute `Action` at time point `T`.
`event(Source,Event,T)`	In	Specifying an event `Event` from `Source` at time point `T`.
`event(request,(ID,Request),T)`	In	A special event, specifying a `Request` with id `ID` at time point `T`.
`event(request,(ID,cancel),T)`	In	A special event, canceling the request with id `ID` at time point `T`.

Fig. 2. Keywords used for communicating between *ROSoClingo* and *clingo*.

the horizon is incremented by one time step. This is realized by assigning `False` to the external atom that identifies the old horizon, followed by the grounding of the transition and query subprograms for the new horizon, and finally, the assignment of `True` to the external atom that identifies this new horizon. Note that the keyword `Fun` represents *clingo*'s data type for function terms, here applied to the external horizon atoms. Finally, `clingoControl` is called again to find a task plan with the new horizon. If an interrupt occurs, the solving process is stopped without *clingo* determining the (un-)satisfiability of the current program and `clingoControl` ends.

The `actionExtractor` identifies actions to be executed during the current logical time point and transforms them into *ROSoClingo* output messages (marked by 4). These messages are then transmitted via the `/ROSoClingo/out` *topic*[6] (marked by 5). It is then the task of the interface layer nodes to transform them into goal requests for the underlying *actionlibs* and to compose a response once the action is executed. The response arrives at the `inputFeeder` component of the *ROSoClingo* node via the `/ROSoClingo/in` topic (marked by 6). The details of how the *ROSoClingo* interface layer interacts with existing ROS components are outlined in Section 2.2.

In contrast to goal requests, messages arriving at the `inputFeeder` component via `/ROSoClingo/out` are transformed into `event` predicates and then incorporated into the existing ASP program as external facts and processed by *clingo*. The keywords of Figure 2 encode the protocol for this (internal) communication between *ROSoClingo* and *clingo*. The second column indicates whether the keyword is an input (in) or part of the output (out) of *clingo*. The (un)successful result of an action may generate new

[6] *Topics* are a named publisher-subscriber communications mechanism for message passing between ROS nodes.

knowledge for the robot about the world (for example, the fact that a doorway is blocked or a new object is sensed).

Once all actions of the current time point report a result the cycle is completed and a new one is initiated, provided there are still actions left to be executed in the task plan. If the task plan is completed *ROSoClingo* waits for new goal requests to be issued.

Finally, it is worthwhile noting that the *ROSoClingo* package is able to support multiple goal requests at a time.

2.2 Integrating with Existing ROS Components

The core *ROSoClingo* node needs to issue commands to, and receive feedback from, existing ROS components. The complexity of this interaction is handled by the nodes at the interface layer (cf. Figure 1). 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 is a need for data type conversions between the individual modules. Turning ROS messages into a suitable set of *clingo* statements therefore requires 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, reasoning about Cartesian coordinates may not be desirable when navigating between named locations such as corridors, rooms and offices. Instead one would hope to reason abstractly about these locations and the relationship between them; for example that the robot should navigate from the kitchen to the bedroom via the hallway.

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. For each existing component that needs to be integrated with *ROSoClingo* there must be a corresponding interface component. We therefore adopt a straightforward message type for messages sent by *ROSoClingo*. This type consists of an assigned name for the robot performing the action and the action to be executed. Note, the addition of robot names allows for the coordination of multiple robots, or multiple robot components, within a single ASP program and to identify the actions performed by each robot or component.

In a similar manner to the *ROSoClingo* output messages, the input messages also consist of a straightforward message type. These messages allow for an interface node to either respond with the success or failure of a *ROSoClingo* action, or alternatively to signal the result of some external or sensory input.

In the scope of the work presented in this paper we implemented an interface to the ROS `move_base` *actionlib*, a standard ROS component for driving a robot. The interface maps symbolic locations with specific coordinates in the environment, e.g. `kitchen` to (12.40,34.56,0.00), and vice versa. The interface node then tracks the navigation task and reports back to the *ROSoClingo* core the success or failure of its task.

3 ASP-based task planning in *ROSoClingo*

The methodology of *ROSoClingo*'s ASP-based approach to task planning is composed of two main activities, viz. *formalizing the dynamic domain* and *formalizing the task as* *a planning problem* in this domain. Each activity involves representing different types of knowledge related to the problem.

The basic principles of this methodology are similar to the general guidelines of representing dynamic domains and solving planning problems in ASP (either it is a direct ASP encoding [9] or an implementation of an action language via ASP [10]). However, since *ROSoClingo* relies upon the multi-shot solving capacities of the *clingo* 4 ASP system [1], the resulting encoding should meet the requirements of the incremental setting, where the whole program is structured as parametrizable subprograms. Multi-shot ASP solving is concerned with grounding and the integration of subprograms into the solving process, and is fully controllable from the procedural side, viz. the scripting language Python in our case. In explaining this process, we first concentrate on the methodology of representing various types of knowledge and later explain the way this knowledge is partitioned into subprograms.

For illustrating the methodology, consider the ASP encoding of a simplified mail delivery scenario, offering a well-known exemplary illustration of action formalisms in robotics [11, 12]: A robot is given the task of picking up and delivering mail packages between offices. Whenever a mail delivery request is received, the robot has to navigate to the office requesting the delivery, pick up the mail package, and then navigate and deliver the item at the destination office. In addition, cancellation requests may happen. If the robot has already picked up the package, it must then return the package to the originating office. Additionally, some of the pathways in the environment may be blocked for some time.

We formalize the dynamic domain by representing the following types of knowledge. Due to space constraints, we provide only representative ASP snippets. One can find the full encoding at [13].

Static knowledge. Time-independent parts of the domain constitute the static knowledge. In view of Section 4, we assume a world instance from the Willow Garage office map and encode this map related information as static knowledge. The following is a snippet from the logic program declaring nodes of waypoints, which are composed of offices, corridors, and open areas, and connections among waypoints.

```
corridor(c1).  corridor(c2).  open(open1).  office(o4).
connection(c3,o4).  connection(c1,open1).  connection(c1,c2).

connection(X,Y) :- connection(Y,X).   waypoint(X) :- corridor(X).
waypoint(X) :- open(X).               waypoint(X) :- office(X).
```

In contrast to static knowledge, dynamic knowledge is time-dependent. In the following program snippets we use the parameter `t` to represent a time point. It is also used as an argument when declaring *clingo* 4's parameterizable subprograms (such as `#program transition(t)`). *ROSoClingo*'s control module incrementally grounds and integrates such programs with increasing integer values for `t`. For instance, the call `ground([("transition",[42])])` grounds the transition subprogram for planning horizon `42`.

In order to specify a state of a dynamic domain, fluents (i.e., properties that change over time) are used. A state associated with a time point `t` is characterized by the fluents captured by atoms of the form `holds(F,t)` where `F` is an instance of a fluent. Figure 3

fluents	`at(W)`	the robot is at waypoint `W`
	`holding(O,P)`	the robot is holding the package `(O,P)`

	`received(request(O,P))`
	`received(cancel(O,P))`
		the robot has received a delivery or cancellation request for a package `(O,P)`

	`blocked(W,W')`	the path between waypoints `W` and `W'` is blocked
actions	`go(W)`	go to the waypoint `W`
	`pickup(O,P)`	pickup the package `(O,P)`
	`deliver(O,P)`	deliver the package `(O,P)`
events	`request(O,P)`	occurs on a request to delivering a package from office `O` to `P`
	`cancel(O,P)`	occurs whenever the request to delivering a package from office `O` to office `P` is cancelled

	`info(blocked(W,W'))`
	`info(unblocked(W,W'))`
		occurs whenever a path between `W` and `W'` is blocked or unblocked

	`value(failure)`	occurs whenever an execution fails

Fig. 3. Fluents, actions, and events used to formalize the domain

lists not only the fluents, but also the actions and exogenous events of the domain. While actions are performed by the robot, events may occur in the dynamic domain without the control of the robot. Actions and events occur within a state of the world and lead to some resulting state. We use the meta-predicates `occurs(A,t)` and `event(E,t)` for stating the occurrence of action `A` and event `E` respectively at time point `t`. We use the following choice rule to allow any action (extensions of `action` predicate includes all actions of the domain) to occur at time point `t`. The upper bound `1` concisely expresses that no concurrent task plans are permitted.

```
{ occurs(A,t) : action(A) } 1.
```

Within the fluents, actions, or events of the domain, we identify each mail package delivery with the pair `(O,P)` consisting of its origin `O` and destination `P`. Although this leads to a simpler encoding, it does limit us to a single delivery from `O` to `P` at a time.

A crucial role in modeling exogenous events is played by *clingo*'s external directives [1]. An `#external` directive allows for, as yet, undefined atoms. To signal external events to the solver, *ROSoClingo* relies upon *clingo*'s library function `assignExternal` that allows for manipulating the truth values of external atoms. For instance, the following rules show how the goal request (based on the signature given in Figure 2) is declared as an external atom and projected into exogenous event `request(O,P)`.

```
#external event(request,(ID,bring(O,P)),t) :- office(O;P), id(ID).
event(request(O,P),t) :- event(request,(ID,bring(O,P)),t).
```

Recall that the first element of the `occurs(Robot,Action,T)` atom (Figure 2) allows for reasoning with concurrent task plans for multi-robot scenarios or for robots with multiple actuators. However, we use `occurs(A,t)` in our case study, since we generate non-concurrent task plans for a single robot. The following rule adds the actuator name.

```
occurs(mailbot,A,t) :- occurs(A,t).
```

Static causal laws. This type of knowledge defines static relations among fluents. They play a role in representing indirect effects of actions. The following rule represents that `blocked` is symmetric and shows how one true `blocked` fluent can cause another `blocked` fluent to be true in a state.

```
holds(blocked(W,W'),t) :- holds(blocked(W',W),t).
```

Dynamic causal laws. Direct effects of actions and events are specified by dynamic causal laws. An action or event occurrence at time `t` can make its effect fluent hold at `t`. Additionally, the occurrence may cancel the perpetuation of fluents. To this end, we use atoms of the form `abnormal(F,t)` to express that fluent `F` must not persist to time point `t`. In robotics, however, action execution failures may occur. Whenever an underlying ROS node fails to perform an action, *ROSoClingo* triggers the `value(failure)` event to signal the execution failure to the encoding. We use atom `executes(A,t)` to decouple the occurrence of action `A` from its effects taking place.

```
executes(A,t) :- occurs(A,t), not event(value(failure),t).
```

This provides us with a concise way of blocking imaginary action effects and thus avoids inconsistencies between the actual world state and the robot's world view. Below are dynamic causal laws for action `go(W)` and event `cancel(O,P)`.

```
holds(at(W),t)      :- executes(go(W),t).
abnormal(at(W'),t) :- executes(go(W),t), holds(at(W'),t-1).
holds(received(cancel(O,P)),t)      :- event(cancel(O,P),t).
abnormal(received(request(O,P)),t) :- event(cancel(O,P),t).
```

In addition, ASP's default reasoning capabilities, together with explicit `executes` and `occurs` statements, pave the way for reasoning with execution failures. For instance, the following rule enables the robot to conclude that the connection to a waypoint is blocked whenever the attempt to navigate to that waypoint fails. (See the third scenario in Section 4 for an illustration.)

```
holds(blocked(W',W),t) :- occurs(go(W),t), not executes(go(W),t),
                          holds(at(W'),t-1).
```

Action preconditions. Action preconditions provide the executability conditions of an action in a state. We use atom `poss(A,t)` to state that action `A` is possible at `t`. Below are preconditions of action `go(W)`. The integrity constraint makes sure that only actions take place whose preconditions are satisfied.

```
poss(go(W),t) :- holds(at(W'),t-1), connection(W',W),
                 not holds(blocked(W',W),t-1).
:- occurs(A,t), not poss(A,t).
```

Inertia. The following rule is a concise representation of the frame axiom.

```
holds(F,t) :- holds(F,t-1), not abnormal(F,t).
```

This completes the formalization of the dynamic domain. Next, we formalize the robot's task as a planning problem.

Initial situation. The following rules represent the initial situation by stating the initial position of the robot.

```
init(at(open3)).
holds(F,0) :- init(F).
```

Goal condition. The following snippet expresses the goal condition. This is the case whenever the robot has no pending delivery request and is not holding any package.

```
goal(t) :- not holds(received(request(_,_)),t),
           not holds(holding(_,_),t).
#external horizon(t).
:- not goal(t), horizon(t).
```

The integrity constraint makes the program unsatisfiable whenever the goal is not reached at the planning horizon. Clearly, this constraint must be removed whenever the horizon is incremented and a new instance with an incremented horizon is added. To this end, we take advantage of the external atom `horizon(t)` whose truth value can be controlled from *ROSoClingo* as shown in Algorithm 1. The manipulation of truth values of externals provides an easy mechanism to activate or deactivate ground rules on demand.

We have mentioned that *clingo* programs are structured into parametrizable subprograms. *ROSoClingo* relies on three subprograms, viz. `base`, `transition(t)`, and `query(t)`. The formalized knowledge is partitioned into these subprograms as follows: `base` contains the time-independent knowledge (static knowledge and initial situation), `transition(t)` contains the time-dependent knowledge (static and dynamic causal laws, action preconditions, and inertia), and finally `query(t)` contains the time-dependent volatile knowledge (goal condition). (See the full encoding at [13].)

4 Case Study

We now demonstrate the application of our *ROSoClingo* package in the mail delivery setting described in the previous section (Section 3). A robot is given the task of picking up and delivering mail packages between offices. Whenever a mail delivery request is received, the robot has to navigate to the office requesting the delivery, pick up the mail package, and then navigate and deliver the item at the destination office.

While the mailbot task is intrinsically dynamic in nature, a secondary source of dynamism is the external environment itself. Obstacles and obstructions are a natural part of a typical office environment, and it is in such cases that the need for high-level reasoning becomes apparent. Our scenario not only highlights the operations of a mail delivery robot in responding to new requests but also shows how such a robot can respond to a changing physical environment.

The office scenario is provided in simulation by the *Gazebo* 3D simulator using an openly accessible world model available for the Willow Garage[7] offices. The robot is a *TurtleBot* equipped with a Microsoft Kinect 3D scanner, which is a cost-effective and well supported robot suitable for small delivery tasks.

From the office environment a partial map has been generated using standard mapping software [14]. This static map is then used as the basis for navigation and robot localization. Furthermore, from this map a topological graph has been constructed to identify individual offices and waypoints that serve as a graph representation for logical reasoning and planning. While this graph has been hand-coded, topological graphs can also be generated through the use of automated techniques [15].

[7] `http://www.willowgarage.com`

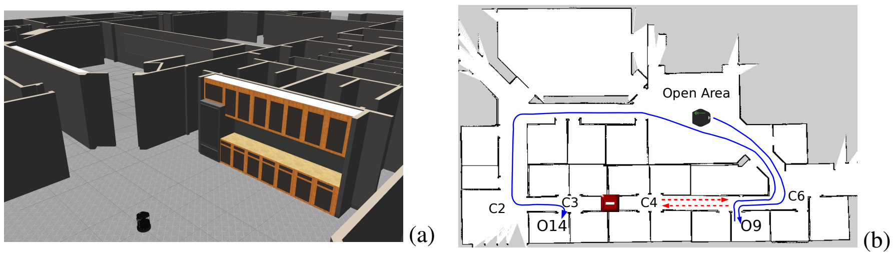

Fig. 4. (a) An office environment for a mail delivery robot, and (b) scenario showing delivery from $O9$ to $O14$, with a blockage dynamically appearing in the corridor between $C3$ and $C4$.

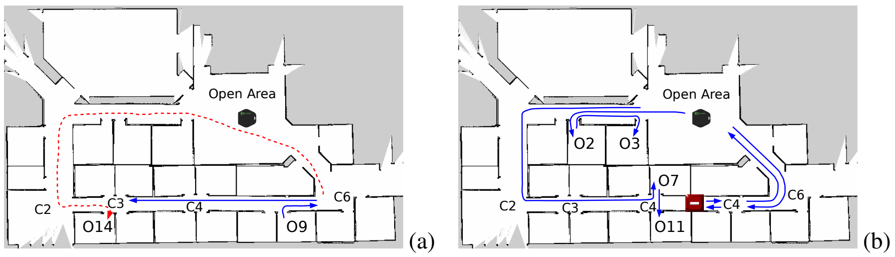

Fig. 5. (a) Scenario showing an obstruction being cleared allowing re-planning for a shorter path through $C4$ and $C3$, and (b) scenario showing adaptive behaviour where changes in the physical environment can affect the order in which tasks are performed.

As previously outlined, *ROSoClingo* provides a simple mechanism for integration with other ROS components, including basic navigation. We further allow for external messages that can be sent to the robot informing it of paths that have been blocked and cleared. In an office environment this can correspond to public announcements, such as work being undertaken in a particular area. Such external messages can also be viewed in the context of the robot receiving additional sensor data.

Finally, as our robot was not equipped with a robot manipulator, item pickup and delivery functionality was simulated by a *ROSoClingo* interface that simply responds successfully to `pickup` and `deliver` action requests.

Scenarios. We consider three scenarios to highlight the behaviour of the mail-delivery robot when it detects and is informed of paths that have been blocked and cleared. In all three scenarios,[8] the robot is initially in the open area shown in Figure 4.

In the first scenario, the robot is told that the corridor is blocked between points $C3$ and $C4$. It is then told to pick up an item from office $O9$ and deliver it to office $O14$. As *ROSoClingo* is able to plan at an abstract level it is able to know that it can move to $O9$ along the optimal route (i.e., via $C6$) but must return through the open area and travel via the corridor point $C2$ in order to reach its destination $O14$. This path is indicated by the solid blue line in Figure 4(b).

[8] The videos of these scenarios are available at `http://goo.gl/g8S5Ky`.

The second scenario (Figure 5(a)) extends that of the first. From $O9$ the robot knows that the path between $C3$ and $C4$ is blocked so it starts to take the long way around as before. However, by the time it reaches $C6$ it has been informed that the blockage has been cleared. This triggers re-planning at the *ROSoClingo* level and the robot is turned around and the shorter path taken through $C4$ and $C3$ to the destination $O14$.

Finally the third scenario shows how dynamic changes to the physical environment can affect the order in which tasks are performed. In this scenario (Figure 5(b)) the robot is first given a task to deliver an item from the office $O7$ to $O11$. While in the vicinity of $C6$ the robot is given a second task to take an item from $O2$ to $O3$. Since it reasons that it is already close to $O7$ the robot continues on with its first delivery task. However, as it progresses past $C4$ the robot detects that the path between $C4$ and $C5$ is blocked. Consequently, the robot has to turn around and take the longer route through the open area. But now offices $O2$ and $O3$ are closer to the robot than $O7$ and $O11$. This causes a change in the robot's task priorities and it swaps the order of tasks, performing the second delivery task first before continuing on with the original.

Discussion. The three mail-delivery scenarios outlined here showcase the adaptive behaviour of the *ROSoClingo* system. The robot is able to respond dynamically to new mail delivery requests while at the same time adapting intelligently to changes in the physical environment. Furthermore, an important property of *ROSoClingo* is that it implicitly performs a form of execution monitoring [16, 17].

Execution monitoring is handled implicitly by *ROSoClingo* because it makes no assumptions about the successful execution of actions. Rather, the *ROSoClingo* interface nodes handle the task of monitoring for the successful completion of actions. This information is then reported back to the reasoner and any failures are handled appropriately.

In fact, because execution monitoring is incorporated directly into the ASP reasoner, *ROSoClingo* can provide for much finer control than is allowed for by traditional systems such as [16]. In particular because execution monitors are specifically designed to deal with anomalous situations, such as action failures, they typically ignore external events that do not result in the failure of the current plan. At first glance, this may seem reasonable. However, in practice it can result in unintuitive and sub-optimal behaviour. For example, in the second mail delivery scenario (Figure 5(a)) the robot replans on the announcement that a blockage has been cleared. Importantly, this re-planning is not triggered as a result of a failure of the current plan, but instead as a recognition of the existence of a better plan. In contrast, because the longer plan is still valid, a traditional execution monitoring based robot would ignore the positive information that the blockage has been cleared and the robot would simply follow the longer route.

Because of *ROSoClingo*'s ability to immediately adapt to new information it bears some resemblance to the Teleo-Reactive programming paradigm of [18]. This goal directed approach to reactive systems is based on *guarded* action rules which are being constantly monitored and triggered based on the satisfaction of rule conditions. However, while Teleo-Reactive systems can provide for highly dynamic behaviour, they typically do not incorporate the complex planning and reasoning functionality of traditional action languages. Hence, in the same way that action language formalisms are rarely applied to highly reactive problem domains, these reactive approaches are rarely applied in problems that require complex reasoning and planning.

However, in constrast to the dichotomy suggested by the difference between these two approaches, many practical real-world cognitive robotic problems do require both highly reactive behaviour and complex action planning. This is highlighted by our mail delivery scenarios where the robot has to undertake its mail deliver tasks while still operating in a dynamically changing physical environment. The successful application of *ROSoClingo* to this task shows that it can be seen as a step towards bridging these two approaches. A robot that incorporates complex reasoning and planning can at the same time adapt to a highly dynamic external environment.

5 Conclusion

We have developed a ROS package integrating *clingo* 4, an ASP solver featuring reactive reasoning, and 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. *ROSoClingo* also makes details of integrating the ASP solver transparent for the developer, as it removes the need to deal with the mechanics of communicating between the solver and external (ROS) components.

Using reactive ASP and *ROSoClingo*, one can control the behaviour of a robot within a single framework in a fully declarative manner. This is particularly important when contrasted against Golog [11] based approaches where the developer must take care of the implementation (usually in Prolog) details of the control knowledge, and the underlying action formalism separately. We illustrated the usage of *ROSoClingo* via a three-fold case-study conducted with a ROS-based simulation of a robot delivering mail packages in the Willow Garage office environment using the *Gazebo* 3D simulator. We showed that ASP based robot control via *ROSoClingo* establishes a principled way of achieving adaptive behaviour in a highly dynamic environment.

This work on *ROSoClingo* opens up a number of avenues for future research. Here we concentrated on the use of *ROSoClingo* for high-level task planning. However *clingo* is a general reasoning tool with applications that extend to other areas of knowledge representation and reasoning such as diagnosis and hypothesis formation. Consequently, an important area for future research would be to consider the use of *ROSoClingo* in these contexts, such as a robot that makes and reasons about the causes of observations in its environment. Another line of future research is to utilize *clingo*'s optimization statements to find optimal task plans when costs of actions are not uniform [19].

Acknowledgments. This work was funded by ARC (DP150103034) and DFG (550/9).

References

1. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Preliminary report. In Leuschel, M., Schrijvers, T., eds.: Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14). Theory and Practice of Logic Programming, Online Supplement. (2014) `http://arxiv.org/abs/1405.3694v1`.

2. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source robot operating system. In: ICRA Workshop on OSS. (2009)
3. Chen, X., Jiang, J., Ji, J., Jin, G., Wang, F.: Integrating NLP with reasoning about actions for autonomous agents communicating with humans. In: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'09), IEEE (2009) 137–140
4. Chen, X., Ji, J., Jiang, J., Jin, G., Wang, F., Xie, J.: Developing high-level cognitive functions for service robots. In van der Hoek, W., Kaminka, G., Lespérance, Y., Luck, M., Sen, S., eds.: Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10), IFAAMAS (2010) 989–996
5. Aker, E., Erdogan, A., Erdem, E., Patoglu, V.: Causal reasoning for planning and coordination of multiple housekeeping robots. In Delgrande, J., Faber, W., eds.: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Springer (2011) 311–316
6. Erdem, E., Aker, E., Patoglu, V.: Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. Intelligent Service Robotics **5**(4) (2012) 275–291
7. Erdem, E., Haspalamutgil, K., Palaz, C., Patoglu, V., Uras, T.: Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'11), IEEE (2011) 4575–4581
8. Andres, B., Rajaratnam, D., Sabuncu, O., Schaub, T.: Integrating ASP into ROS for reasoning in robots: Extended version. Unpublished draft (2015) Available at [13].
9. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
10. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer Academic (2000) 257–279
11. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31**(1-3) (1997) 59–83
12. Thielscher, M.: Logic-based agents and the frame problem: A case for progression. In Hendricks, V., ed.: First-Order Logic Revisited: Proceedings of the Conference 75 Years of First Order Logic (FOL75). (2004) 323–336
13. Potassco website. `http://potassco.sourceforge.net`
14. Grisetti, G., Stachniss, C., Burgard, W.: Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE Transactions on Robotics **23**(1) (2007) 34–46
15. Thrun, S., Bücken, A.: Integrating grid-based and topological maps for mobile robot navigation. In Clancey, W., Weld, D., eds.: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96), AAAI/MIT Press (1996) 944–950
16. De Giacomo, G., Reiter, R., Soutchanski, M.: Execution monitoring of high-level robot programs. In Cohn, A., Schubert, L., Shapiro, S., eds.: Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Morgan Kaufmann (1998) 453–465
17. Pettersson, O.: Execution monitoring in robotics: A survey. Robotics and Autonomous Systems **53**(2) (2005) 73–88
18. Nilsson, N.: Teleo-reactive programs for agent control. Journal of Artificial Intelligence Research **1** (1994) 139–158
19. Khandelwal, P., Yang, F., Leonetti, M., Lifschitz, V., Stone, P.: Planning in action language BC while learning action costs for mobile robots. In: International Conference on Automated Planning and Scheduling (ICAPS). (2014)