**Abstract**

We present REACT!, an interactive tool for high-level reasoning for cognitive robotic applications. REACT! enables robotic researchers to describe robots' actions and change in dynamic domains, without having to know about the syntactic and semantic details of the underlying formalism in advance, and solve hybrid planning problems using state-of-the-art automated reasoners, without having to learn about their input/output language or usage. In particular, REACT! can be used to represent sophisticated dynamic domains that feature defaults, concurrency, indirect effects of actions, and state/transition constraints. It allows for embedding externally defined calculations (e.g., checking for collision-free continuous trajectories) into representations of hybrid domains that require a tight integration of (discrete) high-level reasoning with (continuous) geometric/kinematic/dynamic reasoning. REACT! also enables users to solve planning problems that involve complex goals. Such variety of utilities are useful for robotic researchers to work on interesting and challenging domains, ranging from service robotics to cognitive factories. Furthermore, REACT! provides sample formalizations of some action domains (e.g., robotic manipulation, service robotics, multi-agent path planning), as well as dynamic simulations of plans computed by a state-of-the-art automated reasoner (e.g., a SAT solver or an ASP solver).

**1 Introduction**

As the robotics technology makes its transition from repetitive tasks in highly-structured industrial settings to loosely defined tasks in unstructured human environments, substantial new challenges are encountered. For instance, in order to be able to deploy robotic assistants in our society, these systems are expected to robustly deal with high complexity and wide variability of their surroundings to perform typical everyday tasks without sacrificing safety. Moreover, whenever more than one robotic agent is available in a domain, these robots are expected to collaborate with each other intelligently to share common tasks/resources. The complexity of the tasks and the variability of environment place high demands on the robots' intelligence and autonomy. Consequently, there exists a pressing need to furnish robotic systems with high-level cognitive capabilities.

The multidisciplinary field of robotics is diverse and the technical background of robotics researchers is highly heterogeneous. Even though artificial intelligence (AI) planning and reasoning about actions and change have been studied for decades in the field of computer science, leading to various action description languages, computational methods and automated reasoners, utilization of these outcomes in robotic systems has only recently gained momentum, thanks to increasing demands from new challenging domains, like service robotics applications. However, as many of the robotics researchers trained in diverse engineering aspects of robotics are not familiar with these logic-based formalisms, underlying theoretical AI concepts, and the state-of-the-art automated reasoners, it is still a challenge to integrate high-level automated reasoning methods in robotics applications.

In this paper, we introduce an interactive tool, called REACT!, to fulfill this need in robotics. With REACT!, robotics researchers can describe robots' actions and change in a dynamic domain, without having to know about the syntactic and semantic details of the underlying formalism in advance. Such dynamic domains may be quite sophisticated, allowing defaults, concurrency, indirect effects of actions, and state/transition constraints. They can also solve planning problems, which may involve temporal complex goals, using a state-of-the-art automated reasoner, without having to know about its input/output language or usage. Moreover, while computing a (discrete) plan, some geometric/kinematic/dynamic constraints, for instance on continuous trajectories, can be embedded in domain description, to ensure feasible (e.g., collision-free) plans. Furthermore, thanks to the underlying nonmonotonic logic based formalism of REACT!, commonsense knowledge, in particular exceptions, can be integrated in the domain without causing inconsistencies with them.

REACT! utilizes two sorts of automated reasoners for hybrid planning: SAT solvers (e.g., CHAFF (Meskewicz et al. 2001), MINISAT (Eén and Sörensson 2003), MANYSAT (Hamadi, Jabbour, and Sais 2009)) and ASP solvers (e.g., CLASP (Gebser et al. 2007)). According to SAT (Gomes et al. 2008), the idea is to formalize (in general NP-hard) computational problems as a set of formulas in propositional logic so that the models of this set of formulas characterize the solutions of the given problem; and compute the models using SAT solvers. According to ASP (Lifs-
chitz 2008; Brewka, Eiter, and Truszczynski 2011), the idea is similar; though the problems are represented in a different logical formalism where the formulas (called rules) look different, have a non-monotonic meaning and the models (called answer sets (Gelfond and Lifschitz 1991)) are computed using ASP solvers. Both SAT and ASP have been applied in various real-world applications. For instance, SAT has been used for software and hardware verification (Velev and Bryant 2003) and planning (Kautz and Selman 1992); ASP has been applied to biomedical informatics (Erdem et al. 2011a), space shuttle control (Nogueira et al. 2001), workforce management (Ricca et al. 2012), etc.

REACT! can also utilize continuous-level feasibility checks, such as collision, reachability, graspability and stability checks, by embedding these external computations into high-level reasoning. The feasibility checks can be performed using existing software packages such as OMPL (Sucan, Moll, and Kavraki 2012), MoveIt! (Sucan and Chitta ), OPENRAVE (Diankov 2010), ODE (Smith ) or custom developed software. Similarly, REACT! can integrate background and commonsense knowledge into high-level reasoning, in the style of (Aker et al. 2011a; Erdem, Aker, and Patoglu 2012).

To find a hybrid plan, REACT! follows three steps:

1. REACT! guides the user through an interactive interface to systematically describe the dynamic domain by filling out templates given in natural language. These templates are designed in such a way that the user can declare objects, fluents, actions and their sorts, specify preconditions and effects of actions, and state/transition/concurrency constraints. The user can specify the files for external computations and where to embed these external checks (e.g., in a precondition of an action or in a constraint). Finally, a planning problem is defined to include an initial state, goal conditions and constraints.

2. REACT! automatically transforms the user’s input into the action language C++ (Giunchiglia et al. 2004), which is a nonmonotonic logic-based formalism, specifically designed to represent actions and change (Gelfond and Lifschitz 1998). Depending on the user’s choice of automated reasoner, REACT! also automatically transforms the C++ description into the input language of SAT or ASP solver. These transformations are sound and complete (McCain and Turner 1998; Lifschitz and Turner 1999; Giunchiglia et al. 2004).

3. By automatically interleaving external computations within automated reasoners as specified in the high-level representation of the domain, REACT! computes hybrid plans for the given planning problem using the specified state-of-the-art ASP or SAT solver. It presents the plan to the user in a readable format.

Furthermore, REACT! provides a set of sample cognitive robotics domains (Erdem and Patoglu 2012), including robotic manipulation (Erdem et al. 2011b), cognitive factories (Erdem et al. 2012; 2013b), service robotics (Aker, Patoglu, and Erdem 2012; Aker et al. 2011b; Erdem, Aker, and Patoglu 2012), Tower of Hanoi (Havur et al. 2013) and multi-agent path planning (Erdem et al. 2013a) domains, and dynamic simulation interfaces for execution of hybrid plans in these domains. To implement the dynamic simulations, REACT! utilizes OPENRAVE and Gazebo (Koenig and Howard 2004) simulators.

2 Reasoning about Actions and Change

For an agent to act intelligently in a dynamic domain, one of the essential high level cognitive functions for that agent is reasoning about its actions and the change that is directly or indirectly caused by these actions. For instance, AI planning is one of such reasoning tasks: a robotic agent should be able to autonomously find a sequence of actions to execute, in order to reach a given goal from the initial state she is at. To perform reasoning tasks, an agent should know about which actions she can execute, as well as how these actions change the world. For that, we can describe the actions of the agent in a logic-based formalism so that the agent can autonomously perform reasoning tasks (e.g., find plans) by using an automated reasoner based on logic-based inferences.

On the other hand, representing actions of an agent in a logic-based formalism requires some background in logic as well as the specific representation language. Consider, for instance, a mobile robot that can go from one location to another location, as well as pick and place some boxes in these locations. States of the world can be described by means of three fluents (i.e., predicates whose truth value may change over time): one describing the location of the robot, one describing the locations of objects, another describing whether the robot is holding an object or not. Then, we describe the action of a robot going to a location y, by representing the preconditions (i.e., the robot is not at y) and the direct effects (i.e., the robot is at y). We also describe the indirect effects of actions, state/transition constraints and defaults, like:

Ramifications: If the robot is holding a box b, and the robot goes to some location y, then as an indirect effect of this action the location of the box b becomes y.

State Constraints: Every robot (or box) cannot be at two different locations, and two boxes cannot be at the same location at any state of the world.

Defaults: By default, the boxes can be picked by the robots; exceptions may occur, e.g., if boxes are too heavy for the robots.

Further, we need the commonsense law of inertia: if an action does not change the location of the robot (resp. a box), then the robot’s (resp. a box’s) location remains to be the same.

Figures 1, 2, and 3 show parts of the robot’s domain, in particular the representation of the action of going to a location, in SAT (by a set of clauses), in ASP (by a set of rules) and in C++ (by a set of causal laws), respectively. In these representations, y ranges over locations \( \{L_1, \ldots, L_m\} \), and \( b \) ranges over boxes \( \{B_1, \ldots, B_n\} \). As you can see from these formulations, it is hard to understand which clauses in the SAT formulation describe preconditions and which clauses represent direct effects. The ASP formulation is
Figure 4: A precondition of goto(L1) (i.e., the robot is not already at L1), and a direct effect of goto(L1) (i.e., the robot is at L1).

The robot cannot be at two different locations:
\[ \neg \text{atRobo}(x, t) \lor \neg \text{atRobo}(y, t) \quad (x < y) \]

An object b cannot be at two different locations:
\[ \neg \text{atObj}(b, x, t) \lor \neg \text{atObj}(b, y, t) \quad (x < y) \]

Preconditions of goto(y, t):
\[ \neg \text{goto}(y, t) \lor \neg \text{atRobo}(y, t) \]
\[ \neg \text{goto}(y, t) \lor \neg \text{atObj}(b, y, t) \]

Direct effects of goto(y, t):
\[ \text{atRobo}(y, t + 1) \lor \text{goto}(y, t) \lor \text{atRobo}(y, t) \]
\[ \text{atRobo}(y, t + 1) \lor \neg \text{goto}(y, t) \]

Ramifications:
\[ \neg \text{atObj}(b, y, t) \lor \text{holding}(b, t) \lor \text{atObj}(b, y, t - 1) \]
\[ \text{atObj}(b, y, t) \lor \text{atRobo}(y, t) \lor \text{atObj}(b, y, t - 1) \]
\[ \text{atObj}(b, y, t) \lor \neg \text{holding}(b, t) \lor \neg \text{atRobo}(b, y, t) \]

Figure 3: Describing the robot’s domain in C+

more concise and it is slightly easier to understand each

Preconditions of goto(y):
\[ \text{nonexecutable goto}(y) \text{ if } \text{atRobo} = y \]
\[ \text{nonexecutable goto}(y) \text{ if } \text{atObj}(o) = y \]

Direct effects of goto(y):
\[ \text{goto}(y) \text{ causes } \text{atRobo} = y \]

Ramifications:
\[ \text{caused atObj}(b) = y \text{ if } \text{holding}(b) \land \text{atRobo} = y \]

Figure 2: Describing the robot’s domain in ASP

rule; however, it is still hard to figure out which kind of rules (one with nothing on the left-hand-side of the arrow, or one with some atom) to use for representing what. The C+ formulation is closer to natural language and easier to understand: causal laws of the form nonexecutable a if f describe preconditions \( \neg f \) of an action a; causal laws of the form a causes f describe direct effects f of an action a; and causal laws of the form caused f if g describe ramifications of actions. However, we still need to know about the syntax and semantics of formulas in C+ to formalize a robotic domain.

We would like to enable the robotic researchers to start using automated reasoners to solve various planning problems in dynamic domains with robotic agents, and assist them to have a better understanding of the concepts on reasoning about actions and change, so that they can build/use robots that are furnished with deliberate reasoning capabilities to autonomously perform some tasks. With this motivation, we have built an interactive tool that guides robotic researchers

- to formally represent dynamic domains in a generic way (so they do not have to know about a specific action description language),
- to embed continuous geometric/kinematic/dynamic feasi-
Figure 5: A planning problem.

REACT! helps the users represent a dynamic domain in a logic-based formalism, and solve a hybrid planning problem in this domain using an automated reasoner as follows.

### 3.1 Interactive Intelligent User Interface

REACT! guides the users with an interactive user-interface providing templates in natural language, explanations and examples, so that the users can systematically describe actions and change of a robotic application. For that, they do not have to master any of the underlying formalisms or the reasoners. REACT! also assists users to have an understanding of fundamental concepts described in Section 2.

For instance, the preconditions and direct effects of the action of going to a location are described in REACT! using a template as depicted in Figure 4. As seen in the upper inset, the user describes (in the left part of the user-interface) the following precondition of \textit{goto}(L1): the robot is not already at \textit{L1}. While describing the precondition, the user can use the variables and the object constants (shown on the right part of the user-interface) declared earlier, with the aid of auto-completion. After the user adds this precondition, it is automatically represented in the action language \textit{C+} by the causal: \textbf{nonexecutable goto}(L1) if \textit{atRobo} = \textit{L1}. This causal law is displayed on the right part of the interface, as shown in the lower inset in Figure 4.

Direct effects of actions, ramifications, and state, transition, and concurrency constraints are described similarly by the guidance of the user interface; and the corresponding causal laws are displayed for the information of the users.

Question mark symbols seen on the user-interface provide information about concepts, with examples, for a better understanding. For instance, by clicking the question mark symbols in Figure 4, the user can get information about preconditions and effects of actions, and how they can be specified.

Once the action domain is described and its consistency is verified (otherwise, REACT! gives an error message), REACT! guides the user to specify a planning problem by means of an initial state, goal conditions, constraints, and minimum/maximum plan lengths, as shown in the upper inset in Figure 5. After the problem is specified, the user chooses an automated reasoner and instructs REACT! to compute a feasible plan.

### 3.2 Automated Reasoning with REACT!

As shown in the lower inset in Figure 5, the user can choose one of the state-of-the-art SAT solvers among CHAFF, MINISAT and MANYSAT, or the state-of-the-art ASP solver CLASP as an automated reasoner. Then REACT! automatically transforms user’s specification input via its interactive intelligent interface, into the input language of the reasoner.

If the user chooses a SAT solver, REACT! automatically transforms the causal laws into clauses, similar to the transformation of casual laws in Figure 3 to clauses in Figure 1, as described in (McCain and Turner 1998; Giunchiglia et al. 2004).

Similarly, if the user chooses an ASP solver, REACT! automatically transforms the causal laws into an ASP program, similar to the transformation of casual laws in Figure 3 to ASP program in Figure 2, as described in (Lifschitz and Turner 1999; Giunchiglia et al. 2004).

To implement these sound and complete transformations, REACT! utilizes the tools CCALC (McCain and Turner 1997) and CPLUS2ASP (Casolary and Lee 2011).

Once the user’s specification of action domain is translated into formulas in the input language of the relevant automated reasoner, REACT! instructs the automated reasoner to solve hybrid planning problems using various planners/reasoners (so they do not have to know any specifics of these systems).
to compute a model for these formulas. After the automated reasoner computes a model (if there exists one), REACT! extracts the plan from it, and displays the plan in a readable format, as seen in the right hand side of the lower inset of Figure 5.

### 3.3 Hybrid Planning with REACT!

REACT! allows integration of continuous geometric/kinematic/dynamic constraints (e.g., by calling a motion planner to check whether going to some location by means of a continuous trajectory is feasible without colliding to any static object) as well as temporal constraints (e.g., by defining durative actions and imposing deadlines for completion of tasks) into discrete high-level representation of a domain description, so that the discrete plan computed by REACT! is feasible. Similarly, REACT! can integrate background and commonsense knowledge into high-level reasoning.

Such integrations are possible by means of “external predicates” (McCain 1997; Eiter et al. 2006)—predicates that are not part of the action domain description (i.e., different from fluents and actions) but that are defined externally in some programming language (e.g., C++, Prolog). Integrating geometric/kinematic/dynamic reasoning (in preconditions of actions) and temporal reasoning (in planning problems) are possible thanks to the expressive input language of CCALC.

Essentially, CCALC performs some preprocessing of external predicates, while transforming causal laws into propositional logic formulas. The ASP solver CLASP also supports external predicates.

The lower inset in Figure 6 shows an example of integrating collision checking as part of the preconditions of the action of going to some location \( L_1 \) (from a location \( L_2 \)), by means of an external predicate \( \text{pathExists}(L_1, L_2) \) declared earlier as in the upper inset. By this way, geometric reasoning is “embedded” in high-level representation: while computing a task-plan, the reasoner takes into account geometric models and kinematic relations by means of external predicates implemented for geometric reasoning. In that sense, the geometric reasoner guides the reasoner to find feasible solutions.

External predicates can be defined by a C++ program or a Prolog program. A template for an external predicate definition in C++, to be used for hybrid planning with a SAT solver is presented in Figure 7. For instance, the external predicate \( \text{pathExists}(L_1, L_2) \) is defined using this template: \(<\text{name}>\) describes the predicate name (i.e., \( \text{pathExists} \)), \(<\text{arity}>\) describes the arity of the predicate (i.e., 2), \(<\text{C++ code}>\) implements a geometric reasoning algorithm (or calls a geometric reasoner), and \(<\text{boolean}>\) is true or false.

An interesting example of hybrid planning with temporal constraints is given for the housekeeping domain in (Aker, Patoglu, and Erdem 2012). In this domain, not only an external predicate \( \text{pathExists} \) is used to check geometric feasibility of action of going to some location, but also a second external predicate \( \text{timeEstimate} \) is utilized to estimate the time it will take for the robot to traverse the path. Then, while describing the planning problem, temporal constraints, for instance the constraint that the total time required to com-
plete the plan should be less than a predefined value, can be added to the specification of a planning problem (the upper inset of Figure 5).

Hybrid planning with geometric and temporal constraints and commonsense knowledge has been applied in various domains (e.g., cognitive factories (Erdem et al. 2012; 2013b), service robotics (Aker, Patoglu, and Erdem 2012; Aker et al. 2011b; Erdem, Aker, and Patoglu 2012), robotic manipulation (Erdem et al. 2011b; Havur et al. 2013)) in the spirit of cognitive robotics (Erdem and Patoglu 2012). REACT! allows formulation of all these domains.

3.4 Sample Domains and Simulation Interface of REACT!

To facilitate the use of REACT! for robotic applications and help robotic researchers to gain experience on fundamental concepts in reasoning about actions and change, REACT! provides a set of example domains, including Robotic Manipulation, Service Robotics, Cognitive Factories, Tower of Hanoi and Multi-Agent Path Planning problems.

In each example domain, REACT! provides explanations at each tab (e.g., about the concepts of a fluent, an action, preconditions/effects of an action, ramifications, static/transition constraints, planning problem), so that the user can have a better understanding of systematically representing a dynamic domain.

For each domain, a different set of continuous-level feasibility checks, such as collision checks and path planning queries are provided to be embedded into high-level reasoning as external computations. These feasibility checks can be performed both using existing software OPENRAVE and custom developed software.

Once the user specifies the planning problem and a plan is computed by an automated reasoner of the user’s choice, REACT! also provides a dynamic simulation for an execution of the plan for these domains, using OPENRAVE or Gazebo simulators. For the Tower of Hanoi example, REACT! provides a wrapper to execute the plans on a physical KuKa youBot manipulator through Robot Operating System (ROS). Figure 8 shows a snapshot of a simulation of a plan computed by MiniSAT for a Tower of Hanoi problem, while a movie of its physical implementation can be viewed at http:\\cogrobo.sabanciuniv.edu\\p=690.

4 Conclusions

We have introduced REACT! as an interactive tool for cognitive robotic applications. Significantly reducing the learning time, REACT! lets robotic researchers to concentrate on robotic applications, by enabling them to describe action domains systematically and to solve complex problems relevant to robotics applications using automated reasoners. REACT! not only assists its users to have a better understanding of the concepts on reasoning about actions and change, but also automatically performs sound and complete transformations among different formalisms, embeds external computations into high-level reasoning problems and provides sample formalizations of some action domains, as well as dynamic simulations of plans computed by a selected automated reasoner.

Our ongoing work involves extending the documentation of REACT!, so that REACT! can be made available as an open source program under GPL licence.

References


