On the Design of Symbolic-Geometric Online Planning Systems

Lavindra de Silva[1] and **Felipe Meneguzzi**[2]

[1]University of Nottingham, Nottingham, UK, lavindra.desilva@nottingham.ac.uk
[2]Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, RS, Brazil, felipe.meneguzzi@pucrs.br

Abstract

We describe an abstract multilayered architecture for the organisation of robotic systems that takes into account some of the key functionalities of existing robotic hardware and software in the literature. We demonstrate a concrete instance of the architecture by combining the popular AgentSpeak agent/robot programming language with standard motion planning algorithms. Our work offers some first insights into developing a more formal agent architecture for programming autonomous robots.

1 Introduction

Programming autonomous robotic controllers is an often hard task that requires various types of algorithms ranging from the inverse kinematics used to control actuators and signal processing used to generate sensing information to the highest-level decision making that an autonomous system undertakes to decide which goals are possible and how they can be decomposed. Recent work on robot programming has acknowledged the need to clearly separate concerns in cognitive architectures from the lower level details of the robotic platforms [Ingrand and Ghallab, 2014], thus diverging from earlier work on robot programming [Fleury *et al.*, 1997; Quigley *et al.*, 2009] that often relied on a somewhat ad-hoc process that sees development as a single-step process using virtually the same abstraction throughout the range of robotic functions. Given the complexities of developing autonomous robots, designing the various functions of a robotic system at the right level of abstraction should facilitate development and debugging of complex overall robotic functionality.

In this paper we discuss the main issues in designing symbolic-geometric online planning systems in the form of a desiderata for a multilayered autonomous robot architecture. Our main contribution is a desiderata of the main functions and representational abstractions included at each layer, and linking these layers with existing work in both autonomous agent control and symbolic-geometric planning. We describe the key issues that should be addressed when designing the various layers in a robotic system, enumerate features required to address these issues (in Section 3), and propose one instantiation with an agent architecture (in Section 4) that either addresses specific items in the desiderata or points towards their solution in a modular way using existing techniques (reviewed in Section 2). This is a significant first step in the development of a modular, easily programmable and extensible agent architecture for programming autonomous robots, which we compare with related work in Section 5.

2 Background

2.1 Classical (Symbolic) Planning

Automated planning can be broadly classified into domain independent planning (also called first principles planning) and domain dependent planning. In domain independent planning, the planner takes as input the models of all the actions available to the agent, and a *planning problem specification*: a description of the initial state of the world and a goal to achieve—i.e., a state of affairs, all in terms of some formal language such as STRIPS [Fikes and Nilsson, 1971]. States are generally represented as *logic atoms* denoting what is true in the world. The planner then attempts to generate a sequence of actions which, when applied to the initial state, modifies the world so that the goal state is reached. The planning problem specification is used to generate the state-space over which the planning system searches for a solution, where this state-space is induced by all possible instantiations of the set of operators (i.e. all *ground* instances of operators) using the *Herbrand universe*, derived from the symbols contained in the initial and goal state specifications. Although some planning systems use *lifted* inference (i.e. without generating a fully-grounded subset of the state-space), the vast majority of efficient planners actually does create the set of ground operators, albeit with some strategies to compress it and avoid impossible states. Domain dependent planning takes as input additional domain control knowledge specifying which actions should be selected and how they should be ordered at different stages of the planning process [Nau *et al.*, 1999]. In this way, the planning process is more focused and generates plans faster in practice than with first principles planning.

2.2 Geometric Planning

At the lowest level, geometric planning involves motion planning: looking for a collision-free trajectory that achieves the given goal configuration from the robot's current configuration. In this section we only introduce motion planning. In

general, however, geometric planning may also include additional, higher-level reasoning, such as taking user preferences into account when planning trajectories.

As usual, we use the 3-dimensional world $\mathbb{R}^3$, and define the obstacle region as $O \subset \mathbb{R}^3$. For simplicity, suppose that there is only one rigid robot/body $A \subset \mathbb{R}^3$ in the world.[1] An example of such a body A is the definition of a polygon in the world with each $a \in A$ being a vertex of the polygon. Then, the configuration space C, which is the set of all possible configurations (or poses) of the robot, is a specification of all the possible transformations that could be applied to A. More specifically, a pose $c \in C$ is the tuple $p = \langle x,y,z,h \rangle$, where $(x,y,z) \in \mathbb{R}^3$ and h is the unit quaternion—basically, a four dimensional vector that is used to perform 3D rotations.

We follow the definition of motion planning from Srivastava et al. [2014]. The authors define a motion planning problem as a tuple $\langle C, col, c_I, c_G \rangle$, where $col : C \rightarrow \{true, false\}$ is a function from poses to truth values indicating whether a pose $c \in C$ is in collision ($col(p) = true$) with some object or not, and $c_I, c_G \in C$ are the initial and goal poses. A collision-free motion plan solving a motion planning problem, then, is a sequence $\vec{c} = c_1, \ldots, c_n$ such that $c_I = c_1$, $c_G = c_n$, and for each pose c_i, it is the case that $c_i \in C$ and $col(c_i) = false$.

2.3 BDI-logic Programming

In order to represent the large class of BDI-logic-based [Rao and Georgeff, 1991] programming languages in this work, we adapt the syntax and semantics of the widely extended AgentSpeak(L) [Rao, 1996] programming language. An AgentSpeak agent Ag is a tuple $Ag = \langle Ev, Bel, PLib, Int \rangle$ representing four data structures: a belief base Bel consisting of ground logic atoms following the Prolog convention, on which logic inferences are possible like in Prolog; an event queue Ev to store both external events (environment perception) and internal events (subgoals); a plan library $PLib$ containing BDI plan-rules as defined below; and an intention structure Int containing partially instantiated plan steps derived from the adoption of plan-rules.

A plan-rule $\langle event \rangle : \langle context \rangle \leftarrow \langle body \rangle$ contains three elements: a triggering event $\langle event \rangle$ denoting when a plan-rule is *relevant* for execution; a context condition $\langle context \rangle$ denoting when a relevant plan-rule is *applicable* for execution given an agent's beliefs; and a plan body $\langle body \rangle$ containing a sequence of actions to be executed by the agent. The $\langle event \rangle$ in a plan-rule represents the type of goal the plan-rule handles, as follows: $+\varphi$ or $-\varphi$ where φ is a belief atom, represents that a belief in the agent's belief base has been, respectively added or removed, meaning that the goal is a reaction to events (often) from the environment; $+!\psi$ or $+?\psi$, represent, respectively, an achievement or test goal, meaning that the agent aims to either achieve ψ or test for the validity of ψ in its belief base; and $-!\psi$ or $-?\psi$, represent, respectively, that an achievement or test goal has failed, meaning that either the plan to achieve ψ failed in its execution, or that the belief ψ was not valid. Construct $\langle context \rangle$ is a logical formula, formed using the traditional connectives of conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$), which represents the minimal condition necessary for the plan body in the plan-rule to be executed. Finally, $\langle body \rangle$ is a possibly empty sequence of steps separated by semicolons, where a step is one of the following: *(i)* the execution of an action ϕ in the environment; *(ii)* the adoption of a subgoal $!\psi$ or $?\psi$ (generating an internal event); or *(iii)* the explicit modification of a belief $+\varphi$ or $-\varphi$. Thus, the plan below is an example of a plan-rule in our AgentSpeak-like language:[2]

$$+!move(R,F,T) : canMove(R,F,T) \leftarrow \\ +moving(R); navigate(R,F,T); -moving(R); ?pos(R,T).$$

While a plan-rule is being executed step by step, if an achievement goal such as $!move(robot1, table1, table2)$ is encountered, AgentSpeak then looks up the plan library for a plan-rule that is both relevant and applicable for the goal. Our plan-rule above is relevant for the goal because we can unify $move(R,F,T)$ and $move(robot1, table1, table2)$, by applying substitution $\theta = \{R/robot1, F/table1, T/table2\}$ to the former. Next, if the plan-rule is also applicable with respect to the agent's current beliefs Bel, then the plan body, after applying the substitution to it, is included in the agent's current set of pursued intentions Int. Executing the new intention involves temporarily adding belief $moving(robot1)$ to Bel while the $navigate(robot1, table1, table2)$ action is executed, and then testing the belief base via $?pos(robot1, table2)$ to check whether the navigation was successful. Note that the steps of such plan-rules are executed without any verification of whether the rest of the plan is executable, effectively resulting in an agent that plans and executes online, rather than trying to decompose plan-rules, backtracking when necessary.

3 Desiderata for Symbolic-Geometric Online Planning Systems

The development of autonomous robotic behaviour often divides reasoning into at least two levels of abstraction. At the lowest level, a robot needs to be concerned about the physical constraints of the environment (including static obstructions and moving objects) and its own capabilities (actuators, degrees of freedom, and limitations in sensing), in order to generate motion and sensing plans. At the highest level, a robot needs to reason about goals, user preferences, and temporal constraints. We call these levels, respectively the geometric and the symbolic planning levels, both of which are often embedded within a larger autonomous agent architecture.

There is rich literature about how to represent behaviour at these two levels, such as the work on AgentSpeak(L) [Rao, 1996] and its various descendants [Bordini *et al.*, 2007; Ingrand *et al.*, 1996] and HTN Planning [Nau *et al.*, 1999] on the one hand, and ROS [Quigley *et al.*, 2009] and Genom [Fleury *et al.*, 1997] on the other. In order to have a complete development framework, we need to connect these two levels of reasoning both in terms of data exchanged within their underlying software, but more importantly in terms of translating primitives between abstractions to represent the reasoning occurring at both levels. There are a number of alternatives to do so, which we discuss in terms of

[1] We refer the reader to [LaValle, 2006] for the complete definitions involving non-rigid bodies and those that are not free-floating.

[2] R is short for *Robot*, F is short for *From* and T is short for *To*

the “dominant” abstraction and the direction in which the abstractions are translated.In what follows, we briefly describe three possible approaches, before developing an abstract architecture of robot control that integrates reasoning at various levels, each of which uses a particular abstraction, and communicates abstractions between layers.

First, a naïve approach, as described in Srivastava et al. [2014], consists of “one way” discretisation of the geometric level into the symbolic level, essentially defining an arbitrary level of granularity in the geometric model of the world and creating a symbolic reference to each geometric element. This approach is clearly unfeasible due to explosion in symbolic state space [Srivastava *et al.*, 2014]. Alternatively, a robotic system may use some automated source of symbols that can create an unbounded number of potentially useful symbolic anchors (e.g. create symbols for sets of points in $\mathbb{R}$) as they are generated by events from the geometric world. This is essentially another “one way” translation from geometric to symbolic levels; however, some kind of rule system can be used to decide which geometrical properties (features) are worthy of the high-level reasoner's attention. Finally, one could reason only about a limited set of features predefined at the symbolic level, or “one way” conversion of symbols to geometric elements [Dornhege *et al.*, 2009]. However, this raises the question of how one decides which geometric features should be predefined. Variations of this approach are currently used by most systems that perform symbolic geometric planning; e.g. Srivastava et al. [2014] and Pandey et al. [2012] read and adapt predefined goal poses during planning, but do not generate relevant goal poses from scratch.

Effective mechanisms for autonomous robotic control operate at different levels of abstraction, each having distinct representational requirements [Ingrand and Ghallab, 2014; Ahmadzadeh and Masehian, 2015], which calls for an architecture that separates such levels both conceptually as well as in the implementation. Thus, we envision robotic control organised into a tiered architecture, where each layer represents a separate layer of abstraction that insulates (some) details from the layers above it. This architecture was partially inspired by the levels of integration between robotics and AI by Ingrand and Ghallab [2014]. Our architecture, illustrated in Figure 1, contains five distinct layers with specialised “meta-layers” between some of them.

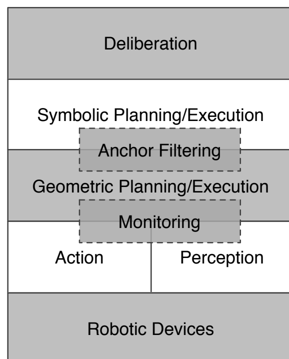

Figure 1: An abstract robot control architecture

Deliberation Level

The deliberation level comprises the highest level of reasoning within an agent, including goal selection [Tinnemeier *et al.*, 2008], meta-level reasoning [Cox and Raja, 2008] and intention scheduling [Waters *et al.*, 2014]. Research on high-level agent architectures such as SOAR [Laird *et al.*, 1987], BDI-based [Rao, 1996] and GDN [Klenk *et al.*, 2013] focuses on such issues, and has generated a breadth of alternatives for high-level robot control while completely abstracting away the details in low-level control.

At the deliberation level, the robot should be able to perform three key functions, all centred around the concept of a *declarative goal* [Winikoff *et al.*, 2002]. First, the agent should be able to select, from a number of possibly conflicting goals, which goals to execute first, and which sets of goals to execute concurrently. For example, an agent may have to decide whether it wants to achieve a goal $at(robot, chargingStation)$ or $at(robot, homePosition)$. This reasoning includes deciding whether each goal is *possible* for the agent, which may include interacting with the layers below it to compute whether a plan to achieve the selected goal exists (possibly performing first principles planning to compute a plan), and whether this plan conflicts with currently executing plans. This interaction with the layers below leads us to the second key function: what strategy to use to make such decisions. In many approaches a simple logic query to an agent's belief base is used in lieu of planning; however, as we shall see in Section 4, some predicates may be linked to much more complex computations. For example, in order to achieve $at(robot, chargingStation)$, an agent may need to consult the symbolic planning layer, which, through its procedural knowledge, eventually consults a predicate $canMove(robot, curr, chargingStation)$, which involves geometric planning. Thus, at the deliberation level, an agent must be able to prioritise which goals are to be computed, and how much computation to spend on them. Finally, even though a plan may be found by the symbolic planning layer, this plan may fail to execute in the real world. This leads us to the third key function, which is to determine how to commit to achieving goals and when to drop a goal by concluding it is not possible. Thus, abstractions at this level include declarative goals [Winikoff *et al.*, 2002], achievement, maintenance and test goals associated with uninstantiated plans or intentions [Cohen and Levesque, 1990], commitments [Meneguzzi *et al.*, 2013], symbolic events from both the environment and the agent itself (such as failed goals) [Rao, 1996] and logic beliefs.

Symbolic Planning and Execution

The symbolic planning and execution layer uses a highly abstracted model of the robot and the environment to perform means-ends reasoning [Meneguzzi and Luck, 2007]. Such reasoning uses the currently perceived world state and some kind of search algorithm to find a sequence of abstracted states that takes the agent into a world in which the currently pursued goals are believed to be true. The main assumption of this level is that the world can be modelled as discrete sets of logic fluents (predicates) and its dynamics in terms of discrete, instantaneous, state transformation operators. These

operators are used either in classical STRIPS-style or hierarchical HTN-style planning, which share symbols with the deliberation level so that goals selected at that level can be used directly as inputs into symbolic planners, and the resulting plans directly manipulated by the deliberation level. When the symbols used by this level of planning refer to the physical (geometric and continuous) world, they are indirectly linked (or *anchored*) to the features manipulated by the geometric planning layer, and ultimately to the physical world outside the robot. Examples of abstractions at this level include: plans and actions [Fikes and Nilsson, 1971] as defined in Section 2.1, hierarchical plans [Nau *et al.*, 1999], non-stationary objects and specific *notable* poses whose relations with objects are either predetermined by a designer or inferable from the environment [Srivastava *et al.*, 2014].

Anchor Filtering

Even if one accepts the loss of detail incurred by discretising an inherently continuous coordinate system, it is not feasible to replicate the entire environment model from the geometric level into the symbolic level [Srivastava *et al.*, 2014], and the resulting explosion in the number of symbolic *anchors* to coordinates renders symbolic planners useless. Thus, any symbolic-geometric planning system needs to carefully evaluate the need to create and maintain every individual *anchor* to entities within the physical coordinate system. A potential approach for this layer include using a reasoner that deals with a high-level initial state and goal, and creating anchors as the agent encounters obstacles in the plans to achieve the goal [Srivastava *et al.*, 2009], and identifying key anchors using some kind of sensor-based filter.

Geometric Planning and Execution

While symbolic planning allows one to intuitively reason about high-level tasks in terms of more specific tasks, and eventually in terms of basic actions, these still "abstract out" the lowest possible level of detail by making certain assumptions about the world. For example, a symbolic operator $move(Robot, From, To)$ might assume that as long as location To is adjacent to location $From$, that the robot at $From$ will be able to navigate to location To. In reality, however, this will not work when certain geometrical characteristics of the robot and the connecting path make the move physically impossible. Combining symbolic planning with geometric planning algorithms used in robotics is therefore crucial to be able to obtain primitive solutions that are viable in the real world.

A necessary component for geometric planning is a representation of the robot's perception of the real world in terms of a 3D world state, which forms part of the input that is needed for planning. Geometric planning then involves standard motion planning within the 3D world, in order to find a trajectory that achieves a given goal pose. In [Pandey *et al.*, 2012], their geometric planner (called a *geometric task planner*) includes functionality that typically resides in the Anchor Filtering layer: their planner takes as input a high-level task instead of a goal pose, maps the task into a set of goal poses at runtime by adapting a predefined set of goal poses, and searches for a viable trajectory for any one of them. Additionally, the mapping process can take into account user-supplied constraints, which may, for instance, preclude goal poses in which an object's "front" does not face the human that is involved in the task. Examples of abstractions at this level include poses, as defined in Section 2.2, numerical constraints and belief distributions about map coordinates (e.g. from SLAM).

Monitoring

In many robotic applications, critical processes in robot control must be monitored continuously in order to ensure suitable reaction times. For instance, a ground robot moving at a certain speed while trying to avoid collisions must stop its movement actuators as soon as it detects an obstacle via short-range sensors such as pressure sensors or sonar. Finally, complex durative actions, such as moving to a specific map coordinate, must use a continuous process integrating monitoring hardware such as an encoder providing odometry readings to the actuators responsible for such motion. Thus, this component connects the Geometric Planning/Execution layer to its counterparts in the Action and Perception layer.

Action and Perception

At its lowest level, *actions* in robotics comprise sequences of actuator and sensor commands with precise timings and monitoring against the tolerances of the robot. Some research has generated design-patterns to program actions at a higher-level, but how these higher-level actions are created and organised seems to be an open problem [Lütkebohle *et al.*, 2011]. *Perception* in robotics comprises either raw sensor data (e.g. a video/image feed and laser scan "point cloud") or some kind of processing over such data (e.g. mapping, positioning, SLAM, and object detection) that allows a robot to build and maintain a model of the environment around it. There are various technologies used to keep the 3D world state up-to-date including tag-based stereo vision systems for object identification and localisation, and Kinect (Microsoft) sensors for localising and tracking humans. These and other action and perception capabilities such as motion control and obstacle avoidance are all included as part of the action and perception layer. These capabilities are typically encapsulated into separate but interconnected modules, which have well-defined interfaces that offer services to the higher layer via communication mechanisms such as "request-response" and "publish-subscribe". In the LAAS [Fleury *et al.*, 1997] architecture for instance, complex functionality such as navigation is derived by suitably linking the inputs and outputs of various independent modules including localisation and path planning. For both action and perception, *granularity* is a key issue: how big must elements be to become eligible to be "named" (e.g. actions for movement or rotation), and how frequently must perceptions from each sensor be obtained.

Robotic Devices

The bottommost layer consists of the control layer for actual hardware control, such as sensor arrays, actuators, and batteries. Control and abstraction at his level is usually performed by specialised software suites with the appropriate "drivers" that convert software commands into specific instructions such as motor pulse sequences, memory address reading to extract images from a CCD sensor, and distance estimation from the ultrasonic sensor pulses. At this level,

control is completely unaware of the larger robot behaviour, and works by simply generating raw readings to be processed and aggregated by the action and perception layers. Examples of software at this level include Programmable Logic Controllers (PLCs) [Antzoulatos *et al.*, 2015] and the most basic topics in ROS [Quigley *et al.*, 2009] such as the various implementations of `cmd_vel` and `sonar`.

4 An Instantiation of our Architecture

We shall now describe an instantiation of our abstract architecture that incorporates motion planning into the AgentSpeak agent programming language, which belongs to the top two layers of our abstract architecture. A fundamental construct that links these two layers with the Geometric Planning/Execution layer is an evaluable predicate, i.e. a predicate that is not evaluated by simply looking up the agent's belief base, but by calling an external procedure, which in our work serves the purpose of searching for a viable trajectory within a geometric 3D world state. Thus, we call such predicates *geometric predicates*. Taking our example from before, we could have the geometric predicate $canMove(R, curr, O)$, which invokes a motion planner to determine whether it is possible for robot R to move from its current pose *curr* to object O (which could, for instance, bind to $table3$), specifically, to a position that is within reachable distance (without further navigation) from O. In this work we use *curr* as a special constant symbol that represents the robot's current pose.

Since geometric predicates are evaluated within a geometric world state, they are implicitly associated with a collection of goal poses, and the predicate holds if and only if there is a viable trajectory that achieves at least one of them. We make this relationship between geometric predicates and goal poses explicit. We do not, however, attempt to automatically infer the goal poses corresponding to a geometric predicate because it is not obvious how this can be done. For instance, it is not obvious how one could automatically determine what goal poses correspond to the suitable "grasps" for differently shaped objects (with different weights) in a domain. Consequently, we assume that a mapping from ground geometric predicates to their corresponding goal poses is supplied by the user. For example, predicate $canMove(robot3, curr, cupboard1)$, might map to the set $\{c_1, \ldots, c_n\}$ of poses, where each c_i corresponds to the robot being at a location from where $cupboard1$ is reachable. Likewise, $canMoveInto(robot1, room2)$ might map to a set of goal poses in which $robot1$ is completely inside $room2$, including the obvious poses where the robot is standing just past the entrance to the room, in its default "rest pose".

Formally, we define the function *map* as follows. Suppose that P is the set of ground predicates obtained from the set of all geometric predicates occurring in the agent, P_s is the set of all predicate symbols occurring in P, O is the set of all constant symbols occurring in P, and that $n = max(\{m \mid m$ is the arity of $p, p \in P\})$. Then, *map* is the partial function

$$map : C \times P_s \times O_1 \times \ldots \times O_n \to 2^C,$$

where C is the configuration space, each $O_i = O \cup \{null\}$, and *null* is a (special) constant symbol. Thus, function *map* is a user-defined "sampling" that includes only the goal poses that "matter" with respect to the current pose $c \in C$ and the given ground geometric predicate. The sampling may, for example, exclude object positions on a table that only differ from other positions by a centimetre. Observe that this mapping belongs to the Anchor Filtering layer of our abstract architecture.

In principle, geometric predicates are evaluated within an "intermediate layer" which is also encapsulated within the Anchor Filtering layer. Our intermediate layer is similar to the one presented in [de Silva *et al.*, 2013a; Srivastava *et al.*, 2014], and actualised via a special evaluable predicate *int*, which is defined as

$$int : P_s \times O_1 \times \ldots \times O_n \to \{true, false\},$$

where P_s, n and each O_i are the same as before. For example, if in the given domain the maximum arity $n = 5$, the agent developer might then invoke the intermediate layer with $int(canMove, robot1, room2, null, null, null)$. We define function $int(p, o_1, \ldots, o_n)$ procedurally as follows. Suppose that c_I is the current pose of the robot, and that S and F are global variables initialised to the empty sequence and empty set, respectively. Then, if there is a pose $c_G \in map(c_I, p, o_1, \ldots, o_n)$, and a collision-free motion plan from c_I to c_G, the function first assigns the motion plan to S, and then returns $true$; otherwise, the function assigns the set of facts describing why there was no trajectory—specifically the obstruction(s) that were involved—to F and returns $false$. This approach keeps motion plans and the need to enumerate over poses transparent from the agent developer.

4.1 Connecting Motion Planning with AgentSpeak

AgentSpeak and related BDI agent programming languages offer some useful, built-in mechanisms that are suitable for incorporating motion planning. In particular, we can "encapsulate" each geometric predicate $p(\vec{v})$ within a unique achievement goal $!e_p(\vec{v})$ as follows. First, we associate the achievement goal with the following two plan-rules:[3]

$$+!e_p(\vec{v}) : true \leftarrow actPass_p(\vec{v}),$$
$$-!e_p(\vec{v}) : true \leftarrow actFail_p(\vec{v}).$$

Since the latter is a plan-rule handling a goal-deletion event, it is only triggered if the former plan-rule fails, i.e. if the precondition of the ground action $actPass_p(\vec{v})\theta$,[4] which involves geometric planning, is not applicable. Moreover, as per the semantics of goal-deletion events, once the latter rule finishes executing, the associated achievement goal $!e_p(\vec{v})\theta$ still fails. This semantics is desirable in order to, before failing, compute and include the beliefs/facts relating to why the failure occurred. The second step in our encapsulation is to define actions $actPass_p$ and $actFail_p$ via the following operators:

$$actPass_p(\vec{v}) : int(p, \vec{v}) \leftarrow body_\top \; ; \; post_\top$$
$$actFail_p(\vec{v}) : \neg int(p, \vec{v}) \leftarrow body_\bot \; ; \; post_\bot,$$[5]

[3] We use $\vec{v}$ to denote a vector of distinct variables, and $\vec{t}$ to denote a vector of (not necessarily distinct) variables and/or constants.

[4] where θ is the relevant substitution that was computed for the variables in $\vec{v}$ as per the operational semantics of AgentSpeak

[5] Actually, the last parameters of $int(p, \vec{v})$ must be zero or more *null* constant symbols, based on the arity of predicate *int*. We have adapted the definition of an operator from [Sardiña *et al.*, 2006].

where construct $body_\top$ is associated with code that executes the motion plan S computed by $int(p,\vec{v})\theta$; $body_\bot$ is associated with an empty procedure, as nothing needs to be executed if there was no trajectory found while evaluating the precondition; and constructs $post_\top$ and $post_\bot$ obtain and apply the set of symbolic facts concerned with, respectively, the pose that resulted from executing $body_\top$, and the "reasons" why there was no trajectory while evaluating the precondition, i.e. the set F computed by $int(p,\vec{v})\theta$. Observe that the second operator above confirms that $\neg int(p,\vec{v})\theta$ still holds; this is done just in case there was a relevant change in the environment after $int(p,\vec{v})\theta$ was last checked, causing $int(p,\vec{v})\theta$ to now hold, in which case there are no failure-related facts to include.

We assume that the procedure associated with $body_\top$ always succeeds, and check whether the action was successful by explicitly testing its desired goal condition. This is exemplified by the $!move(R,F,T)$ achievement goal in Section 2.3, where $?pos(R,T)$ checks whether the $navigate(R,F,T)$ action was successful. One property of our encapsulation is that looking for motion plans and then executing them and/or applying the associated symbolic facts are one atomic operation, so no other step can be interleaved between those steps. Thus, our approach ensures that a motion plan found while evaluating an action's precondition cannot be invalidated by an interleaved step while the action is being executed.

Once all geometric predicates occurring in the agent have been encapsulated as described, we may then use their corresponding achievement goals from within AgentSpeak plans. However, since preconditions of actions and plan-rules cannot mention achievement goals, those encapsulating geometric predicates are instead (WLOG) placed as the first steps of plan bodies. A desirable feature of this approach is that it enables such achievement goals to be ordered so that the ones associated with the most computationally expensive geometric predicates are checked only if the less expensive ones were already checked and they were met. This is similar to the approach taken by [de Silva *et al.*, 2013a; Kaelbling and Lozano-Pérez, 2013], who allow ordering predicates occurring in preconditions.

4.2 Computing Symbolic Facts

The facts applied to the (symbolic) world state by functions $post_\top$ and $post_\bot$ can include both domain-dependent as well as domain-independent facts, computed by the Anchor Filtering layer by using the layers beneath. Function $post_\top$ derives symbolic facts corresponding to the current (geometric) pose of the robot using function $fct : C \to 2^{\mathbf{P}}$, where the infinite set of ground predicates $\mathbf{P} = \{p(\vec{v})\theta \mid p(\vec{v})\theta$ is a ground instance of $p(\vec{v}), \theta$ is a substitution, $p(\vec{t}) \in P^{all}\}$, where P^{all} is the set of all predicates occurring in the agent. In words, fct is a mapping from poses to a set of sets of facts, where the latter only mentions predicate symbols that occur in the agent, but might also mention constant symbols (objects) that do not occur in the agent. This leaves room for discovering new objects "on the fly", which were not previously known to the agent. For example, after executing a trajectory, the agent might find itself in a position from where a previously unidentified bottle can be seen. Once it is identified as a bottle by the Action and Perception layer, it might be assigned a new symbol such as $redBot1$, and associated with the new (symbolic) facts $bottle(redBot1)$ and $near(redBot1, bottle1)$.

Other examples of the kinds of domain-dependent facts that might be computed by $post_\top$ are ground instances of the predicates $inside(Rm,R), upright(O), visible(O,R)$, and $reachable(O,R)$, where O is an object, R is a robot, and Rm is a room. Predicate $visible(O,R)$ holds if and only if O is visible from R, and $reachable(O,R)$ holds if and only if it may be possible for robot R to reach O without navigating from the current position [de Silva *et al.*, 2013a]. The facts corresponding to all of these predicates are easy to compute: e.g. $visible(O,R)$ simply involves checking whether there is a line-of-sight from robot R to object O, and $reachable(O,R)$ involves checking whether the volume covered by extending the robot's arms and torso, with respect to all their degrees of freedom, overlaps with object O in 3D space.

Unlike function $post_\top$, the facts computed by $post_\bot$ "describe" why a trajectory did not exist for the associated geometric predicate. One example of such a fact is $\neg reachable(bottle1, robot1)$, indicating $bottle1$ is (definitely) not reachable from $robot1$ without navigating—because a trajectory did not exist from $robot1$ to $bottle1$ (which does not necessarily mean that there is an obstructing object). Other examples of such facts, inspired by [Srivastava *et al.*, 2014], are $obstructsSome(cup3, bottle2, robot1)$, indicating that $cup3$ obstructs at least one trajectory from $robot1$ to $bottle2$, and $obstructsAll(cup3, bottle2, robot1)$, indicating that $cup3$ obstructs all the trajectories from $robot1$ to $bottle2$.[6] These facts could be exploited by the agent system, e.g. to plan to move obstructing objects out of the way.

5 Related Work

Our approach to incorporating geometric planning into a BDI-style agent system is inspired by existing work in the literature, particularly [de Silva *et al.*, 2013a; de Silva *et al.*, 2013b; de Silva *et al.*, 2014; Srivastava *et al.*, 2014; Kaelbling and Lozano-Pérez, 2013]. In [Kaelbling and Lozano-Pérez, 2013] the authors present an approach to interleaving acting with planning in the belief space of agents. They define specific fluents to represent abstractions of more complex geometrical properties, and associate those fluents with procedures which are evaluated at runtime, based on the most recent geometric world state. This is similar to how we use evaluable predicates to test whether certain geometrical properties hold at runtime. On the other hand, while we use a traditional operator representation, and indeed a traditional BDI agent programming language, they use a special purpose representation that is nonetheless well suited for their specific planning and execution framework (e.g. additional "let" and "exists" constructs that can be included in preconditions).

De Silva et al. [2013a; 2014] rely on an intermediate layer between symbolic planning and geometric reasoning that is similar to the one that we have presented: both involve performing motion planning within traditional preconditions, computing the resulting symbolic facts, and then applying

[6]For simplicity, these predicates do not represent which actions' trajectories (e.g. "pick" versus a "place") were obstructed.

them to the symbolic world state. However, while their work focuses on interleaving geometric reasoning with symbolic planning, this paper focuses on interleaving geometric reasoning with acting in the real world. This makes the details of the two approaches different: e.g. while our approach accounts for discovering and managing references to new objects in the environment when acting in the real world, this is not an issue in their work. Our work also shares some similarities with the interface to symbolic and geometric planning presented in Srivastava et al. [2014], as discussed in Section 4.2. Their approach involves obtaining a classical planning solution for the given planning problem, and then checking if there is a viable trajectory for each adjacent pair of actions in the solution; however, they do not focus on interleaving geometric reasoning with execution, as we do here.

While the systems discussed below also do not address the issue of combining geometric planning with *execution*, they nonetheless do fit within the abstract architecture presented in Section 3. Like de Silva *et al.* [2013a], the work of Lagriffoul *et al.* [2012] presents an integration that combines a variant of the SHOP [Nau *et al.*, 1999] HTN planner and a specialised path planner, in which the geometric planner backtracks when an action being planned at the symbolic level is not applicable. This allows their system to reconsider the geometric choices that were made, in order to "fix" the symbolic plan being synthesised. The symbolic planner of Erdem *et al.* [2011] guides a motion planner—invoked via evaluable predicates—toward a continuous collision-free trajectory, failing which the symbolic planning problem is adjusted to take the cause of failure into account when replanning. The integrations presented in [Dornhege *et al.*, 2009; Gaschler *et al.*, 2015] also rely on evaluable predicates in order to perform geometric reasoning. In these works, evaluable predicates are used for, e.g. trajectory planning, reachability analysis, collision checking, and inverse kinematics.

Unlike the works described, Asymov [Cambon *et al.*, 2004] is a combined task and motion planning framework in which it is the geometric planner that makes use of the symbolic planner (and its domain), rather than the other way around. This involves obtaining heuristics from the symbolic planner when choosing roadmaps during geometric search. The work of [Plaku and Hager, 2010] is similar to the Asymov approach, where a symbolic planner guides a sampling-based motion planner's exploration of the continuous space, and the latter returns utility estimates that are used to improve the symbolic planner's guidance in the next iteration.

6 Conclusion

This paper has summarised a diverse range of functionalities present in existing robotic systems at various levels of abstraction, and separated them into the distinct layers of an abstract architecture, partly inspired by the levels of integration in [Ingrand and Ghallab, 2014]. We then presented an instantiation of certain interesting elements of our architecture, by combining the AgentSpeak agent programming language with motion planning. In particular, we described a suitable interface between the two types of system, showed how some existing AgentSpeak constructs could be exploited to incorporate motion planning, and briefly classified the kinds of symbolic facts that could be computed in the geometric layer and passed back to the symbolic. In the future we intend to formalise the integration of AgentSpeak and motion planning, as well as develop a corresponding implementation.

Acknowledgements

Felipe thanks CNPq for support within process numbers 306864/2013-4 under the PQ fellowship and 482156/2013-9 under the Universal project programs.

References

[Ahmadzadeh and Masehian, 2015] Hossein Ahmadzadeh and Ellips Masehian. Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization. *AIJ*, 223(0):27 – 64, 2015.

[Antzoulatos *et al.*, 2015] Nikolas Antzoulatos, Elkin Castro, Lavindra de Silva, and Svetan Ratchev. Interfacing agents with an industrial assembly system for "plug and produce". In *AAMAS*, pages 1957–1958, 2015.

[Bordini *et al.*, 2007] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007.

[Cambon *et al.*, 2004] Stéphane Cambon, Fabien Gravot, and Rachid Alami. A robot task planner that merges symbolic and geometric reasoning. In *ECAI*, pages 895–899, 2004.

[Cohen and Levesque, 1990] Phillip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *AIJ*, 42(2-3):213–261, 1990.

[Cox and Raja, 2008] Michael Cox and Anita Raja. Metareasoning: A manifesto. In *Procs. AAAI 2008 Workshop on Metareasoning: Thinking about Thinking*, 2008.

[de Silva *et al.*, 2013a] Lavindra de Silva, Amit Kumar Pandey, and Rachid Alami. An interface for interleaved symbolic-geometric planning and backtracking. In *IROS*, pages 232–239, 2013.

[de Silva *et al.*, 2013b] Lavindra de Silva, Amit Kumar Pandey, Mamoun Gharbi, and Rachid Alami. Towards combining HTN planning and Geometric Task Planning. In *RSS Workshop on Combined Robot Motion Planning and AI Planning for Practical Applications*, 2013.

[de Silva *et al.*, 2014] Lavindra de Silva, Mamoun Gharbi, Amit Kumar Pandey, and Rachid Alami. A new approach to combined symbolic-geometric backtracking in the context of human-robot interaction. In *ICRA*, pages 3757–3763, 2014.

[Dornhege *et al.*, 2009] Christian Dornhege, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel. Semantic attachments for domain-independent planning systems. In *ICAPS*, pages 114–121, 2009.

[Erdem *et al.*, 2011] 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. In *ICRA*, pages 4575–4581, 2011.

[Fikes and Nilsson, 1971] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ*, 2(3-4):189–208, 1971.

[Fleury *et al.*, 1997] S. Fleury, M. Herrb, and R. Chatila. Genom: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *IROS*, pages 842–849, 1997.

[Gaschler *et al.*, 2015] Andre Gaschler, Ingmar Kessler, Ronald P. A. Petrick, and Alois Knoll. Extending the Knowledge of Volumes Approach to Robot Task Planning with Efficient Geometric Predicates. In *ICRA*, 2015.

[Ingrand and Ghallab, 2014] Félix Ingrand and Malik Ghallab. Robotics and artificial intelligence: A perspective on deliberation functions. *AI Commun.*, 27(1):63–80, 2014.

[Ingrand *et al.*, 1996] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*, pages 43–49, 1996.

[Kaelbling and Lozano-Pérez, 2013] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Integrated task and motion planning in belief space. *IJRR*, 32(9-10):1194–1227, 2013.

[Klenk *et al.*, 2013] Matthew Klenk, Matt Molineaux, and David W Aha. Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence*, 29(2):187–206, 2013.

[Lagriffoul *et al.*, 2012] F. Lagriffoul, D. Dimitrov, A. Saffiotti, and L. Karlsson. Constraint propagation on interval bounds for dealing with geometric backtracking. In *IROS*, pages 957–964, 2012.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: an architecture for general intelligence. *AIJ*, 33(1):1–64, 1987.

[LaValle, 2006] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, USA, 2006.

[Lütkebohle *et al.*, 2011] Ingo Lütkebohle, Roland Philippsen, Vijay Pradeep, Eitan Marder-Eppstein, and Sven Wachsmuth. Generic middleware support for coordinating robot software components: The task-state-pattern. *Journal of Software Engineering for Robotics*, 2(1):20–39, 2011.

[Meneguzzi and Luck, 2007] Felipe Meneguzzi and Michael Luck. Composing high-level plans for declarative agent programming. In *Proceedings of the Fifth Workshop on Declarative Agent Languages*, pages 115–130, 2007.

[Meneguzzi *et al.*, 2013] Felipe Meneguzzi, Pankaj R. Telang, and Munindar P. Singh. A first-order formalization of commitments and goals for planning. In *AAAI*, 2013.

[Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *IJCAI*, pages 968–973, 1999.

[Pandey *et al.*, 2012] A.K. Pandey, J.-P. Saut, D. Sidobre, and R. Alami. Towards planning human-robot interactive manipulation tasks: Task dependent and human oriented autonomous selection of grasp and placement. In *IEEE International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, pages 1371–1376, 2012.

[Plaku and Hager, 2010] E. Plaku and G.D. Hager. Sampling-based motion and symbolic action planning with geometric and differential constraints. In *ICRA*, pages 5002–5008, 2010.

[Quigley *et al.*, 2009] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[Rao and Georgeff, 1991] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *KR*, pages 473–484, 1991.

[Rao, 1996] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Proceedings of the 7th MAAMAW*, volume 1038 of *LNCS*, pages 42–55. Springer-Verlag, 1996.

[Sardiña *et al.*, 2006] Sebastian Sardiña, Lavindra de Silva, and Lin Padgham. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *AAMAS*, pages 1001–1008, 2006.

[Srivastava *et al.*, 2009] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Challenges in finding generalized plans. In *ICAPS Workshop on Generalized Planning: Macros, Loops, Domain Control*, 2009.

[Srivastava *et al.*, 2014] Siddharth Srivastava, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. In *ICRA*, pages 639–646, 2014.

[Tinnemeier *et al.*, 2008] Nick A.M. Tinnemeier, Mehdi Dastani, and John-Jules Ch. Meyer. Goal selection strategies for rational agents. In *Languages, Methodologies and Development Tools for Multi-Agent Systems*, volume 5118 of *LNCS*, pages 54–70. Dastani, M. and El Fallah Seghrouchni, A. and Leite, J. and Torroni, P., 2008.

[Waters *et al.*, 2014] Max Waters, Lin Padgham, and Sebastian Sardiña. Evaluating coverage based intention selection. In *AAMAS*, pages 957–964, 2014.

[Winikoff *et al.*, 2002] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & Procedural Goals in Intelligent Agent Systems. In *KR*, pages 470–481, 2002.