2011 IEEE/RSJ International Conference on
Intelligent Robots and Systems
September 25-30, 2011. San Francisco, CA, USA

Parameterizing Actions to have the Appropriate Effects

Lorenz Mösenlechner and Michael Beetz
Intelligent Autonomous Systems Group
Karlstr. 45, D-80333 München
{moesenle,beetz}@cs.tum.edu

***Abstract*— Robots that are to perform their tasks reliably and skillfully in complex domains such as a human household need to apply both, qualitative and quantitative reasoning to achieve their goals. Consider a robot whose task is to make pancakes, and part of the plan is to put down the bottle with pancake mix after pouring it on the pan. The put-down location of the bottle is heavily under-specified but has a critical influence on the overall performance of the plan. For instance, when it places it at a location where it occludes other objects, the robot cannot see and grasp the occluded objects anymore unless the bottle is removed again. Other important aspects include stability and reachability. Objects should not flip over or fall. A badly chosen put-down location can "block" trajectories for grasping other objects that were valid before and can even prevent the robot from reaching these objects. In this paper, we show a lightweight and fast reasoning system that integrates qualitative and quantitative reasoning based on Prolog. We demonstrate how we implement predicates that make use of OpenGL, the Bullet physics engine and inverse kinematics calculation. Equipped with generative models yielding pose candidates, our system allows for the generation of action parameters such as put down locations under the constraints of the current and future actions in real time.**

I. Introduction

Mobile robots as shown in Figure 1 that are to perform complex activity such as setting a table or preparing a meal have to make many critical decisions that influence the overall performance of their actions. Action sequences that are generated by a symbolic planner or imported from WWW knowledge [1] are under-specified because they are purely symbolic. If the robot is making pancakes, a part of its plan is to first pour the pancake dough on the pancake maker and then put down the bottle with dough. This put down action lacks many parameters, for instance the exact location where to place the bottle. A random place does not suffice. Rather, we want to place the bottle while keeping it reachable because we might need it later and we do not want to "block" grasping trajectories of other objects in the scene. The bottle should be stable, i.e. we do not want to place it on other objects and with upright orientation. We also want to keep all objects that are also required in our plan visible because we only can grasp visible objects.

The space of possible parameterizations of actions when performing complex activity such as making pancakes is extremely big and correct solutions are constrained in many ways. This includes geometric and dynamic properties of the world, i.e. physics, but also common sense knowledge. On the other hand, many solutions can be found that are all correct but some lead to higher performance. Finding a good

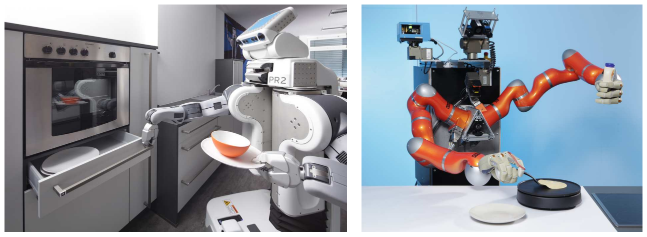
Fig. 1. Mobile robot platforms performing complex activity in a human household domain.

set of parameters provides the opportunity for optimizations that cannot be done in purely symbolic systems.

In this paper, we describe a lightweight reasoning system that is capable of generating parameterizations such as locations in real time to reliably performing actions. The system not only has deep symbolic knowledge about the objects in the environment through an ontology but also the ability to make realistic predictions of the outcome of an action in a *specific environment configuration*. Our system combines features of symbolic reasoning engines based on depth-first search with the power of physics engines that allow for accurately inferring the sub-symbolic properties of the world. This is achieved by implementing predicates that reason about the continuous world. These predicates first set up the world representation of a physics engine, simulate it for a small amount of time, perform collision detection and calculate bindings and the truth value of the predicates from the world state after simulation. More specifically, our system combines a symbolic Prolog based reasoning engine with the Bullet physics engine [1] and an OpenGL based rendering engine to make inferences about visibility. In combination with generative models for locations, it allows for reasoning about the stability of scenes, visibility of objects in the scene and reachablility of objects and to generate solutions for poses that satisfy such queries.

In the remainder of this paper, we proceed as follows. First we give an overview of related work. Then we give a detailed description of our reasoning system and explain how predicates for reasoning about stability, reachability and visibility are implemented. Finally, we show how we integrate the system in an executive on a real robot.

II. Related work

The idea of integrating qualitative and quantitative reasoning is not new. An early system is explained in [2] which

[1] www.bulletphysics.org

Authorized licensed use limited to: STAATS U UNIBIBL BREMEN. Downloaded on April 22,2025 at 09:33:53 UTC from IEEE Xplore. Restrictions apply.

describes the combination of numerical thermal and fluid dynamics simulation and symbolic properties. Planners have been developed that are combining geometric and symbolic planning, such as [3], [4] and [5]. An interesting aspect of [5] is that the authors integrate geometric reasoning into a symbolic planner by resolving the applicability of actions using geometric reasoning. This approach is similar to implementing predicates based on qualitative reasoning components as presented in this article. But in addition to pure geometric reasoning about collisions and reachability, our system is also capable of inferring stability and visibility of scenes. [3] even integrates dynamics through a physics engine. [6] integrates a general purpose simulation into a tableaux based reasoning engine while ignoring other important aspects such as perspective taking, visibility or reachability. [7] demonstrates a system for visibility reasoning and perspective taking in human robot interaction. The work shown in [8] is based on executing complete action sequences in simulation and recording all important aspects. Prolog is then used to perform reasoning on the recorded execution scenarios. Using a complete simulation environment allows for very accurate results, though being limited by the accuracy of the physics engine. But this approach is computationally very expensive because all aspects of action execution, including the control loops running on a robot, are simulated. In contrast, our system is designed to be lightweight and only simulates what is required to answer a query. This makes our system applicable for on-line decision making during action execution on a robot. The applicability of physics engines to predict action consequences have been demonstrated in [9] where the authors show that accurate simulation results can be achieved by tuning simulation parameters. The placement of objects based on spacial relations specified in natural language has been presented in [10]. There, the authors use predefined areas, so called spacial tags to resolve properties such as *on* or *under*.

III. The Reasoning System

In this section, we will explain how we integrate sub-symbolic reasoning mechanisms that allow for inferring physical properties of scenes, visibility and reachability into Prolog's reasoning engine. The idea is to implement *predicates* in the Prolog engine that are resolved by using the bullet physics engine, OpenGL and similar mechanisms. These predicates are calculated on-demand, whenever the Prolog engine needs to proof them. In contrast to imperative systems that set up a world, simulate it and monitor the simulation to search for specific (most often hard-coded) conditions, predicates allow for combining quantitative reasoning with symbolic knowledge provided by, for instance, systems like Knowrob [11]. Using Prolog not only gives a well-defined and powerful interface for reasoning but also allows to solve complex queries such as: *"find all objects that are required for making pancakes and locations to place them"*. We apply the system for decision making in pick-and-place tasks that are performed in a human household domain. The system needs to reason about occlusions because occluded objects cannot be seen and therefore cannot be grasped. It needs to find locations that allow for fast calculation of short grasping trajectories and putting down an object should keep the world stable. The example above also contains queries for purely symbolic information, (the objects that are required to make pancakes) which is resolved by using a purely symbolic knowledge base.

To explain how our reasoning system works, let us consider the simple query "find a location for a mug on a plate that is stable". To solve such a query, the system performs the following steps:

1) Copy the current belief state
2) Generate a pose for the mug that is on the plate based on a generator pattern
3) Move the mug to the generated pose in the copied belief state
4) Check if the scene is stable
5) If the scene is unstable, backtrack and continue with step 2

The corresponding prolog query of our example can be formulated as follows:

?- *similarWorld(W,* W_{tmp}*), object(*W_{tmp}*, plate, Plate), object(*W_{tmp}*, mug, Mug), randomPoseOn(Plate, Mug, P), assertPose(Mug, P), stable(*W_{tmp}*).*

Please note that we need to work on a copy of the belief state because we want to keep the initial belief state unchanged to allow for further inferences. Prolog processes the goals (i.e. predicates) successively and tries to prove them under the set of the current variable assignments. Each step might add new variable bindings to this set. If more than one solution is valid, Prolog creates a choice-point and continues by trying to prove the next goal. If a goals fails, i.e. all possible variable bindings that satisfy it are contradicting the set of current bindings, Prolog backtracks to the innermost choice-point, selects a new solution from the set of possible variable assignments and continues the process. In the example above, the predicate *randomPoseOn* creates an infinite number of poses that are all on the *Plate*, places the mug at the generated position and continues searching until the predicate *stable* holds. Then it yields all variable bindings as one solution. All choice-points are kept open until the query is closed which allows the user to successively generate more solutions until all possible variable bindings are processed. Please note that the query above will yield an infinite number of solutions (or run infinitely if there are no solutions) because the number of generated poses is not limited. This can easily be solved by generating the (infinite) list of possible poses and then taking just the first n solutions.

One of the most important parameters for successfully and reliably performing pick-and-place tasks are the locations of the objects the action is performed on. We need to place objects such that task-specific constraints on visibility, stability and reachability are fulfilled. To reason about these aspects of the world, we define six predicates:

- *contact(W,* O_1*,* O_2*)* holds if the two objects O_1 and O_2 are in contact in the world *W*.

Authorized licensed use limited to: STAATS U UNIBIBL BREMEN. Downloaded on April 22,2025 at 09:33:53 UTC from IEEE Xplore. Restrictions apply.

- *stable(W, O)* holds if all forces on object O are canceled out, i.e. it does not move in the corresponding world W.
- *visible(W, P, O)* holds if the object O is visible from pose P in the world W.
- *occluding(W, P, O_1, O_2)* holds if object O_2 is occluding the object O_1 when the camera is at pose P.
- *reachable(W, R, O)* holds if the robot R can reach the object O in the world configuration W.
- *blocking(W, R, O, B)* unifies B with the list of objects that might be blocking a grasp for object O.

Table I shows a list of predicates that we define in our reasoning system. Their semantics and implementation are discussed in the remainder of this article.

Predicates to interact with the world database	
world(W)	Unifies W with a bullet world database
similarWorld(W_1, W_2)	Holds if W_1 and W_2 contain the same objects at the same locations
assertObject(W, T, N, P)	Asserts an object O of type T and name N at pose P
retractObject(W, O)	Retracts an object O from the world database
object(W, N, O)	Asserts that O is an object in world W with name N
pose(O, P)	Unifies P with the pose of object O
assertPose(O, P)	Moves the object O to pose P
Predicates to interact with the robot model	
linkPose(R, N, P)	Unifies P with the pose of the robot link named N of robot model R
Predicates to reason about stability	
contact(W, O_1, O_2)	Holds if objects O_1 and O_2 are in contact in world W
stable(W, O)	Holds for all objects O that are stable in world W
stable(W)	Holds if all objects are stable in world W
Predicates to reason about visibility	
visible(W, P, O)	Holds if the object O is visible from pose P
occluding(W, P, O_1, O_2)	Holds if the object O_2 occludes object O_1 when the camera is at pose P
Predicates to reason about reachability	
reachable(W, R, O)	Holds if the robot R can reach object O
reachable(W, R, O, S)	Holds if the robot R can reach object O with S indicating the right or the left arm
blocking(W, R, O, B)	Unifies B with the list of objects that might be blocking a grasp for object O

TABLE I

PREDICATES USED TO PERFORM PHYSICS BASED INFERENCES

Spatial reasoning in symbolic reasoning systems is most often based on qualitative representations of space. These systems make inferences on qualitative units of space, for instance *behind*, *above* or *on*. Using predicate calculus on a real robotic system, however, requires mechanisms to translate symbolic spatial descriptions into actual locations in Cartesian space. While verifying if a specific position corresponds to a symbolic equivalent is relatively easy, inferring a solution for a pose that is part of the solution of a Prolog query is rather difficult. To calculate the truth value of a *leftOf* predicate given two objects, comparing the coordinates of both objects is normally sufficient. Finding a good position for one of the objects that satisfies the predicate, on the other hand, is a complex task because an infinite number of valid solutions are possible. In this paper, we propose a set of task-specific generative models that randomly generate poses. Objects in the (continuous) world of the physics engine are positioned according to the sampled poses and the system infers the assignment of all other (symbolic) variables for a respective query. If the query holds, the solution can be used as a parameterization for an action. Otherwise, a new set of pose samples is generated until a solution is found or a maximal number of samples is reached.

A. Physics Engine Integration

Our system is based on the Bullet physics engine. Bullet is an industry strength physics engine mainly used in computer games. Its main components are a collision engine for performing collision checks between objects, an engine for rigid body dynamics and an engine for soft body dynamics. Bullet provides reasonable performance while sacrificing accuracy. The reason is that in its main application domain, computer games, high accuracy is less important than nicely looking effects. But on the other hand, for integrating physics into symbolic reasoning, good performance is more important than an extremely high degree of realism. We do want to know *that* an object flips over in a scene while we surely cannot predict how and where it falls. Even with a more accurate physics engine, sensor noise would prevent exact prediction. Although Bullet is not perfectly accurate, [9] show that parameters can be tuned to improve accuracy.

We use the physics engine to calculate bindings and truth values for the predicates *contact* that holds for all pairs of objects that are in contact in a specific scene configuration and *stable* that holds if a specific scene is stable.

In the current system, we only use rigid body dynamics in combination with Bullet's collision engine. Adding soft-body dynamics for simulating cloths or even pancakes will be added in the future.

To infer the predicates from a scene, we first need to generate a scene from a symbolic representation of the world. For the robot and most of the objects it manipulates, we already have 3D models that can be used for rigid body simulation. The view of Bullet's world in the Prolog engine is similar to a Prolog fact database where information can be asserted or retracted. To interact with the bullet world, we define the predicate *world(W)* that unifies the variable W with a world database object that corresponds to a bullet world. To assert objects, we define the predicate *assertObject(W, Type, Name, Pose)* that creates a new object of type *Type* at the specified pose if an object with the same name does not exist already. Please note that we explicitly pass a world object. This allows us to deal with different worlds in the same reasoning process. The implementation of the predicates is relatively simple: to create a world, we create a new instance of the corresponding Bullet data type. *assertObject* inserts single rigid bodies such as plates or mugs but also adds more

Authorized licensed use limited to: STAATS U UNIBIBL BREMEN. Downloaded on April 22,2025 at 09:33:53 UTC from IEEE Xplore. Restrictions apply.

complex objects such as the robot or a model of the kitchen we perform our actions in that consist of several rigid bodies.

Most often, the interpretation of a Prolog query takes an existing world and changes it by adding or moving objects. For instance, to assert that the world is stable, we simulate it for a short period of time. To not change the original world database, we define the predicate *similarWorld(*W_1*,* W_2*)* that creates an exact copy of the current belief state and its dynamic state and binds it to a variable.

As already mentioned, we implement the predicates *contact(W,* O_1*,* O_2*)* and *stable(W,* O_1*)* based on the physics engine. Please note that it is not necessary to use temporal calculus for this kind of reasoning. The predicate *contact* is implemented based on Bullet's collision engine. The engine calculates for all pairs of colliding objects a collision manifold that contains the points where the objects are colliding. The predicate corresponds to three different functionalities, depending on the variables that are bound when the reasoning engine tries to prove it. The variable W always has to be bound to a bullet world object since it corresponds to our central database containing the scene. The two objects can either be both bound, only one of the two can be bound or both are unbound. The former case results in a collision check. If only one of the variables is bound, solutions are generated for each pair of colliding objects where one object corresponds to the value of the bound variable. If none of the object variables is bound, solutions are generated for every pair of colliding objects.

While the *stable* predicate has to simulate the dynamics of the current scene for a short period of time, it is still not necessary to introduce temporal calculus here as long as the world state is not changed by the predicate. This is achieved by copying the belief state before starting the simulation. With Bullet, it is relatively easy to store the complete state of a world in order to restore it later or copy it. The system iterates over all rigid bodies and copies their mass, and the dynamic state, i.e. the pose, the velocity and the torque vector of the object. In addition a reference to its collision object has to be stored. Apart from rigid bodies, some properties of the world, such as the gravity vector, are copied, too. The object variable of the *stable* predicate can either be bound or unbound when the inference engine tries to prove it. If bound, it leads to a verification of stability of the respective object. The world is first copied and simulation for a short period of time is performed on the copied world (0.5 seconds turned out to be sufficient). Then the pose of the object is compared to its pose in the original world object. If the object moved, i.e. if its pose changed, the predicate fails. Otherwise, it succeeds. It the object is unbound, the implementation first stores the poses of all objects and starts simulation. Then, only the objects for which the pose didn't change are used to create solutions for the predicate.

Figure 2 shows two unstable scenes where a mug is standing on the edge of a plate and a plate placed on a mug. The figure shows the initial scene and after running physics simulation for 0.5 seconds.

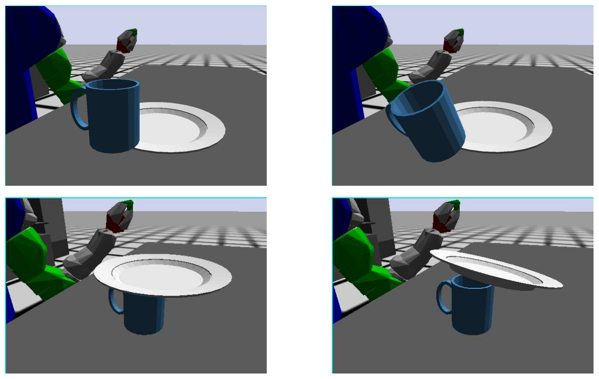
Fig. 2. Unstable scenes in the initial configuration (left) and after simulating for 0.5 seconds. In the top row, a mug is standing on the edge of a plate, in the bottom row, the plate has been placed on a mug.

B. Visibility Computation

Inferring the visibility of objects in a specific camera frame when the robot is standing at a specific position is an important part of our reasoning system. When placing objects, the system can infer if other objects will become invisible. When searching for an already known object, poses for the robot can be generated from where the object should not be occluded by other objects. Consider an example where the robot places an object on the table. Then it drives around the table and performs an action such as putting down another object. Our system then allows to infer a location that allows to detect the previously placed object, i.e. that allows to generate a pose where the original object is visible from the other side of the table.

For making inferences on the visibility of objects, we define the two predicates *visible(W, P, O)* and *occluding(W, P,* O_1*,* O_2*)*, both implemented based on OpenGL and off-screen rendering.

OpenGL allows to render into an invisible frame buffer. To infer visibility of objects, we render the scene with each object in a different color and count how many pixels of every object are rendered. Since every object is rendered in a different color, we can directly map pixels on the screen to the corresponding object. To calculate visibility, we compare the number of pixels that belong to a specific object to the number of pixels that the object should have.

More specifically, we render the scene three times, as shown in Figure 3. First, we place the camera at the pose the scene should be rendered from and point it directly on the object. This leads to the approximate number of pixels that should be visible if the object is not occluded. Centering the object in the camera to calculate the total number of pixels belonging to the object is necessary since an object could be just at the boundary of the camera and only visible by parts. In this case, our system should infer that, for instance, the object is visible by only 50 % but not occluded by any other object.

The second rendering step draws the object with the camera positioned correctly. This gives us a "mask" that can be used to infer which objects are occluding it.

Authorized licensed use limited to: STAATS U UNIBIBL BREMEN. Downloaded on April 22,2025 at 09:33:53 UTC from IEEE Xplore. Restrictions apply.

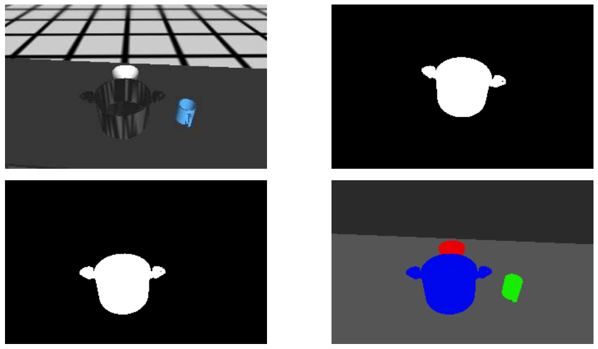

Fig. 3. Generated images to infer the visibility (and occluding objects) of the pot. Top left: rendered scene from robot perspective; top right: the pot centered; bottom left: only the pot; bottom right: the complete scene, every object has unique colors. The pot is (partly) occluding the bowl.

The final rendering operation renders the complete scene, including the robot and the environment. We then iterate over the whole scene, counting all pixels that belong to the object we want to calculate visibility for. Further, we count all pixels that correspond to object pixels in the mask generated in rendering step two but that do not belong to the object.

To calculate if an object is visible or not, we take the ratio of pixels that belong to the object and are visible in rendering step three. If it is greater than a certain threshold (e.g. 90 %), we consider the object to be visible. Occluding objects are then calculated by mapping the pixels that do not belong to the object but are in the mask of rendering step two to the corresponding objects by using the pixel color.

For using the predicate *visible*, the variables W and P need to be bound while O can be either bound to an object or left unbound. If bound, the procedure described above has to be executed only once. If unbound, we first unify every object that is asserted in W. Then we prove the predicate *visible* for each of the objects, which generates a solution for each object that is visible. Please note that the reason for requiring the pose variable P to be bound is that generating a pose from which an object is visible required sampling in at least a 5-Dimensional space, defined by the location of the robot's base and the values of the joints along the kinematic chain from the base of the robot to the camera. These parameters constrain the pose of the camera and therefore, these values needed to be sampled to generate pose candidates for the camera frame.

The predicate *occluding*(W, P, O_1, O_2) holds if the object O_2 is occluding the object O_1 in the world W when the scene is viewed from pose P. The predicate requires the variables W and P to be bound but can create solutions for the two objects. When both objects are bound, the procedure as explained above for inferring occluding objects is used to prove that O_2 is part of the set of occluding objects. If O_2 is unbound, one solution for each object that is a member of the set of occluding objects is created by binding that object to O_2. Finally, if O_1 is unbound, every object that is known in the world is checked for occlusion.

C. Reachability Reasoning

Manipulation is one of the most important but also one of the most difficult tasks when performing actions in a human household domain. The robot needs to stand at a location that allows it to reach the objects it is supposed to grasp and grasping the object requires expensive motion planning as well as grasp planning.

Our system is designed to generate approximate solutions that permit a high probability to successfully execute an action. It does not include motion planning or grasp planning as, for instance, described in [12], which we consider as computationally too expensive to be integrated in a reasoning system that is used to infer parameters at run-time. Rather, we want to approximately decide if a location is good enough to grasp an object or if a grasp might be blocked by other objects. We define the predicate *reachable*(W, R, O) that holds if an object O is reachable by the robot R in the world W and the predicate *blocking*(W, R, O, B) that yields the list of blocking objects bound to the variable B. The current implementation of the two predicates only requires the variable W to be bound.

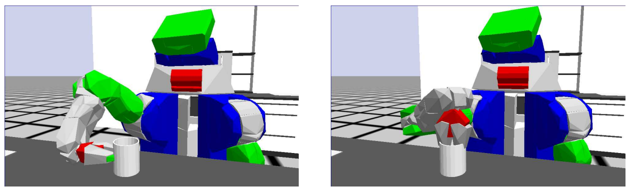

Fig. 4. Side and top grasp used to infer reachability and blocking objects.

To infer reachability and blocking objects, we define two "grasps", a side and a top grasp, as shown in Figure 4. Reachability is resolved by trying to find a solution for inverse kinematics for any of the two grasps. Please note that this is an approximation of reachability because we do not take into account self-collisions, collisions with the table and collisions with other objects. In the rare case where no collision free motion plan can be found, the executive system that is actually controlling the robot has to handle this failure at run-time.

To infer an approximate solution for blocking objects, instead of searching for just one IK solution, we generate a fixed number of solutions with different initial (seed) states for both of the two grasps. This results in different arm configurations for reaching the same pose with the end-effector as shown in Figure 5. The system then performs collision checking between the robot and the environment by using the predicate *contact* and builds the set of all contacting objects excluding the object to be reached.

D. Generative Models

In order to infer parameters such as locations we apply generative models that create pose candidates which are then verified by Bullet and OpenGL based reasoning. Consider the example of proving the query "find a location to place two mugs on a plate in order to carry the objects all together under the constraint that the mugs must not occlude each other". The query for just proving that the above condition holds can be stated in Prolog as follows:

Authorized licensed use limited to: STAATS U UNIBIBL BREMEN. Downloaded on April 22,2025 at 09:33:53 UTC from IEEE Xplore. Restrictions apply.

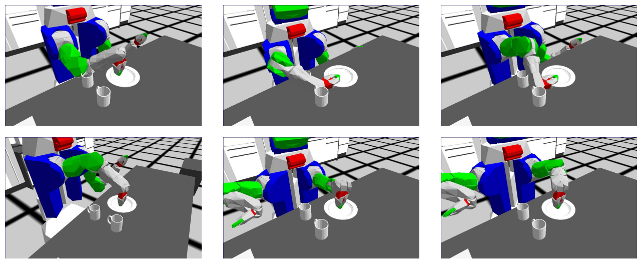

Fig. 5. Inverse kinematic solutions that collide (top row) and that do not collide (bottom row) for different arms and different grasps.

?- *world(W), object(W, mug$_1$, Mug$_1$), object(W, mug$_1$, Mug$_2$), object(W, plate, Plate), stable(W, Mug$_1$), stable(W, Mug$_2$), stable(W, Plate), linkPose(W, Robot, 'camera_frame', Cam), visible(W, Cam, Mug$_1$), visible(W, Cam, Mug$_2$).*

To search for poses that satisfy the expression above, we generate a (possibly infinite) sequence of pose candidates for the two mugs, assert their pose in the world, i.e. move the objects to the corresponding locations, and try to prove the query. In other words, we apply sampling to generate candidates of solutions and then use the physics and rendering based parts of the reasoning engine to check if the respective candidate satisfies the query. Please note that in the approximately continuous world representation that we use for physics based reasoning, proving that a query cannot be satisfied is extremely hard if not impossible. The current system has no means of proving that there exists no solution. For our domain of finding parameters for programs that are executed on a robotic system, i.e. in a domain with a high degree of uncertainty in many different components, we assume that there exists no solution if a sufficient number of pose samples has been processed without finding a valid solution.

Generative models "guide" the search for solutions. The better the candidates that are generated are, the faster a valid solution can be found. That means the more domain knowledge we put into generating poses, i.e. the more we reduce the dimensionality of the task, the faster is the inference. Pose generation is based on the computation of a probability distribution function that is then used to sample pose candidates. All constraints that apply to a certain pose are compiled to a corresponding functions and the combination of these functions yields the actual probability distribution. Constraining factors include the *on* relation, the maximal distance between the camera and the object where it is still possible to detect the object and the maximal reachability radius of the robot, defined by the length of the robot's arms.

In the current system, we are mainly interested in placing objects *on* other objects under additional constraints. That gives us the opportunity to constrain several dimensions of the 6-dimensional space of poses. We set the z coordinate (which is pointing up in the world) of an object such that the object is directly on top of the underlying object. In other words, we use the bounding boxes of the two objects to calculate the value of the z coordinate of the top object. In addition, we use the projection of the supporting object on the x-y-plane to constrain sampling in x- and y- direction since positions that are not above the supporting object will never satisfy a corresponding query. For most objects that are manipulated, rotation can be ignored, too. Objects such as plates, glasses, mugs or a bottle with pancake mix are either rotationally symmetric or approximately rotationally symmetric. The rotation of other objects, such as silverware that is used to set a table, is normally constrained by the task context and can be inferred by using a common sense knowledge base. We can therefore reduce the space we need to draw samples from to a two-dimensional plane with restricted extent as shown in Figure 6 (left). Given a fixed robot pose, reachability implies that the object needs to be in a certain range around the robot. Thus, a corresponding probability distribution for one arm can be approximated by a circle. If we want to prefer poses that are closer to the robot, we can overlap the circle with a two-dimensional Gaussian. All constraints that our system supports can be compiled into a corresponding probability distribution. To combine them, we discretize all functions using a specific grid size, for instance 2.5cm, multiply all values with the same coordinates and normalize the resulting discrete function to have a sum of one. Then we interpret it as a probability distribution and draw samples from it. The complete distribution of a location on the table that might be reachable by the robot is shown in Figure 6 (right).

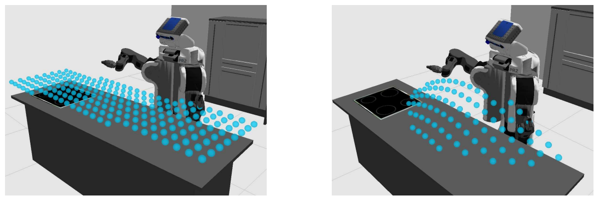

Fig. 6. Uniform distribution of locations on the table (left) and for locations on the table that can be reached by the robot (right).

IV. Integration and Application Example

As already mentioned, we apply our system in a domestic environment to find locations where the robot should stand while performing actions and where to put objects. The executive we use is based on the Cram Plan Language [2]. The most important key aspect of the system is that control programs contain annotations that define the semantics of the respective program part in first-order logic [13]. Actions are parameterized by designators, symbolic descriptions of the respective parameter that are resolved based on the context of the action and the capabilities of the robot. For instance, we describe a location to stand in order to grasp a cup with the designator

(a location ((to reach) (obj Cup1)))

Cup1 is an object designator describing the cup. The designator describing a location where the cup could be put down on the counter is specified as follows:

(a location ((on plane) (name counter) (for Cup1)))

We can add more constraints to the designator, for instance that the object *PancakeMix* should not become invisible and

[2]www.ros.org/wiki/cram_language

Authorized licensed use limited to: STAATS U UNIBIBL BREMEN. Downloaded on April 22,2025 at 09:33:53 UTC from IEEE Xplore. Restrictions apply.

that the cup *Cup1* should be reachable by the robot. Such a designator is specified as follows:

(a location ((on plane) (name counter) (for Cup1)
(visible PancakeMix) (in-reach Cup1)))

Designators, in particular location designators, are the entry point for the reasoning system shown in this article. Designators always consist of key-value pairs and can directly be translated into a Prolog query. The corresponding Prolog query for the designator above is defined as follows:

?- *object(W, counter, Counter)*, *object(W, cup1, Cup1)*,
object(W, mondamin, PancakeMix), *object(W, pr2, Robot)*,
linkPose(Robot, 'kinect_optical_frame', Kinect),
randomPoseOn(Counter, Cup1, P),
stable(W, Cup1), *visible(W, Kinect, PancakeMix)*,
reachable(W, Robot, Cup1)

An assignment for the variable *P* that is generated by this query then satisfies the properties of the original designator and is used as a solution. If the runtime system figures out that the solution is invalid, i.e. while satisfying all constraints the solution still leads to an error, a new solution is generated. This is possible since sampling yields an infinite number of solutions and the designator implementation keeps the Prolog query open.

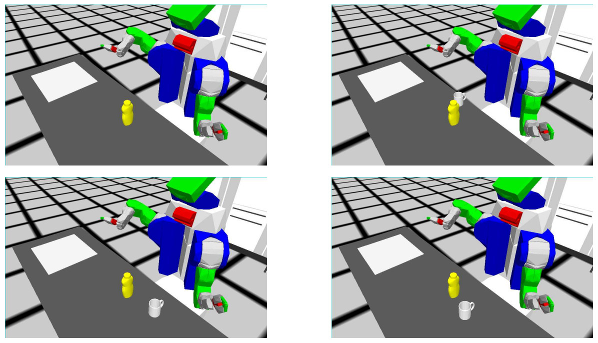

Fig. 7. Initial scene and three different solutions for the query "find a pose for the cup with the pancake mix being visible and the cup being reachable".

The run-time required to find a solution for complex queries such as that one above depends on many factors. The most important one is the generative model that is used to sample pose candidates. If a distribution as shown in Figure 6 is used for sampling, the mean run-time is 1.3 seconds on an Intel Core2 Duo with 2.4 GHz. That indicates that the system is suitable for real-time decision making in a highlevel executive. Motion planning and action execution take much more time in most cases.

V. Conclusions and Future Work

In this paper, we have presented a Prolog based reasoning system that integrates qualitative and quantitative reasoning. We have shown how we implement predicates that use the Bullet physics engine, OpenGL and generative models for sampling to infer variable bindings and truth values. The system enables a robot to find solutions for parameters such as the location where to stand or where to place objects in a robust and flexible way.

We have integrated the system in our executive that runs on our two robots to infer locations where to place objects and where to stand. But the range of possible applications of the system is much wider. One example is using the system for prediction. It allows for predicting which objects should be visible and even can generate point clouds of how the scene *should* look like. In combination with reasoning about executed plans [13], we will implement a powerful projection mechanism that allows for predicting the outcome of a plan and inferring the reason why a plan failed. Performance might be improved by using *smart objects* [14] that provide semantic information on how the robot can interact with them in order to reduce the search space for possible solutions.

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

References

[1] M. Tenorth, D. Nyga, and M. Beetz, "Understanding and Executing Instructions for Everyday Manipulation Tasks from the World Wide Web." in *IEEE International Conference on Robotics and Automation (ICRA).*, 2010, pp. 1486–1491.

[2] K. Forbus and B. Falkenhainer, "Self-explanatory simulations: An integration of qualitative and quantitative knowledge," 1990, pp. 380–387.

[3] S. Zickler and M. Veloso, "Efficient physics-based planning: sampling search via non-deterministic tactics and skills," in *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*. Richland, SC: IFAAMAS, 2009, pp. 27–33.

[4] S. Ruehl, Z. Xue, T. Kerscher, and R. Dillmann, "Towards automatic manipulation action planning for service robots," in *KI 2010: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, R. Dillmann, J. Beyerer, U. Hanebeck, and T. Schultz, Eds. Springer Berlin / Heidelberg, 2010, vol. 6359, pp. 366–373.

[5] C. Dornhege, M. Gissler, M. Teschner, and B. Nebel, "Integrating symbolic and geometric planning for mobile manipulation." in *IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR)*, 2009. [Online]. Available: http://www.informatik.uni-freiburg.de/~ki/papers/dornhege-etal-ssrr09.pdf

[6] B. Johnston and M. Williams, "Comirit: Commonsense Reasoning by Integrating Simulation and Logic," in *Artificial General Intelligence 2008: Proceedings of the First AGI Conference*. IOS Press, 2008, p. 200.

[7] L. Marin, E. A. Sisbot, and R. Alami, "Geometric tools for perspective taking for human-robot interaction," in *Mexican International Conference on Artificial Intelligence (MICAI 2008)*, 2008.

[8] L. Kunze, M. E. Dolha, E. Guzman, and M. Beetz, "Simulation-based temporal projection of everyday robot object manipulation," in *Proc. of the 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Yolum, Tumer, Stone, and Sonenberg, Eds. Taipei, Taiwan: IFAAMAS, May, 2–6 2011.

[9] E. Weitnauer, R. Haschke, and H. Ritter, "Evaluating a physics engine as an ingredient for physical reasoning," in *Proceedings of the Second international conference on Simulation, modeling, and programming for autonomous robots*, ser. SIMPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 144–155.

[10] B. Coyne and R. Sproat, "Wordseye: an automatic text-to-scene conversion system," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 487–496.

[11] M. Tenorth and M. Beetz, "KnowRob — Knowledge Processing for Autonomous Personal Robots," in *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2009, pp. 4261–4266.

[12] D. Berenson, R. Diankov, K. Nishiwaki, S. Kagami, and J. Kuffner, "Grasp planning in complex scenes," in *IEEE-RAS International Conference on Humanoid Robots*, 2007.

[13] L. Mösenlechner, N. Demmel, and M. Beetz, "Becoming Action-aware through Reasoning about Logged Plan Execution Traces," in *Submitted to the IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2010.

[14] M. Kallmann and D. Thalmann, "Modeling objects for interaction tasks," in *Proc. Eurographics Workshop on Animation and Simulation*, 1998, pp. 73–86.

Authorized licensed use limited to: STAATS U UNIBIBL BREMEN. Downloaded on April 22,2025 at 09:33:53 UTC from IEEE Xplore. Restrictions apply.