2017 IEEE International Conference on Robotics and Automation (ICRA)
Singapore, May 29 - June 3, 2017

Improving the Reliability of Service Robots in the Presence of External Faults by Learning Action Execution Models

Alex Mitrevski[‡], Anastassia Kuestenmacher, Santosh Thoduka, and Paul G. Plöger

***Abstract*— While executing actions, service robots may experience external faults because of insufficient knowledge about the actions' preconditions. The possibility of encountering such faults can be minimised if symbolic and geometric precondition models are combined into a representation that specifies how and where actions should be executed. This work investigates the problem of learning such action execution models and the manner in which those models can be generalised. In particular, we develop a template-based representation of execution models, which we call δ models, and describe how symbolic template representations and geometric success probability distributions can be combined for generalising the templates beyond the problem instances on which they are created. Our experimental analysis, which is performed with two physical robot platforms, shows that δ models can describe execution-specific knowledge reliably, thus serving as a viable model for avoiding the occurrence of external faults.[1]**

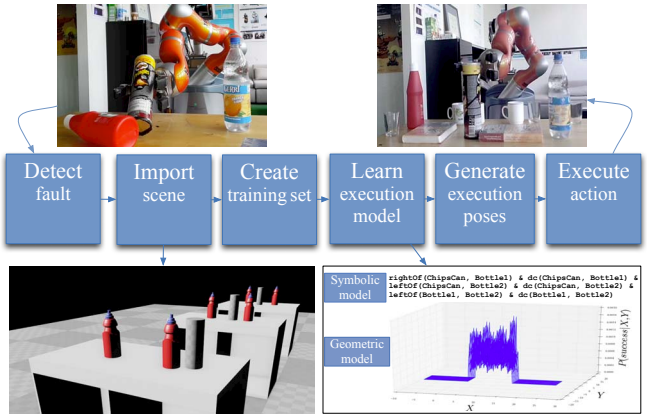


Fig. 1: Learning and execution schema

I. INTRODUCTION

A. Motivation

While acting, service robots may encounter events that lead to incorrect execution of a given set of actions. Such events are sometimes caused by defective internal robot components, in which case they are commonly referred to as *internal faults* [1]. Unfortunately, the performance of a robot can degrade even if all internal components are working perfectly well, which happens when anomalous events are caused by factors that lie outside the robot's physical system boundary. We refer to such anomalies as *external faults* [2].

External faults come in various forms, but this work only deals with those that are caused by physical object properties; faults that are caused by external agents, such as humans or other robots, are not considered. But even under this constraint, anticipating every event that could lead to an external fault is nearly impossible. This can be illustrated with the simple example shown in Fig. 1: a scenario[2] in which a robot has to put a chips can on a table that has two bottles on it, such that the desired outcome is to have the can standing at an arbitrary position on the table.[3] While releasing the can, the robot may experience various external faults: the passive bottles may be knocked down because of a collision, the manipulated can may fall down on the floor or might not stand upright due to an incorrect release orientation or an inappropriate release height, and so forth.

Manually created external fault models could alleviate some of the problems in this and many similar scenarios in which an agent's performance suffers due to incomplete execution knowledge; however, they are often insufficient for dealing with external faults because (i) the list of events that could affect the execution success could go on indefinitely and (ii) external faults are almost always scenario-specific. This suggests that a feasible strategy for increasing an agent's reliability in the presence of such faults is a learning by experimentation mechanism that allows the agent to estimate how to perform a given action autonomously. In this paper, we use a simulation for solving the experimentation problem.

B. Focus of This Paper

Assuming that our agent has a plan-based system and that the desired action outcomes are specified in advance, we address the following questions: (i) how to represent action execution knowledge effectively; (ii) how a given execution model can be applied when dealing with various environment configurations and distinct objects; and (iii) how to integrate model updates into the learning architecture.

In particular, we define action execution via so-called δ models, which are object- and task-dependent templates that combine symbolic and geometric knowledge obtained in well-defined configurations with a fixed number of passive objects. We show that δ models can be created using pairwise object relations and that successive symbolic and geometric

*This work was supported by the B-IT foundation

†The authors are with the Department of Computer Science, Hochschule Bonn-Rhein-Sieg, Sankt Augustin, Germany
<aleksandar.mitrevski, anastassia.kuestenmacher, santosh.thoduka, paul.ploeger>@h-brs.de

‡Corresponding author

[1]The code accompanying this paper can be found at https://github.com/alex-mitrevski/delta-execution-models

[2]Referred to as *bottle scenario* throughout the paper.

[3]Henceforth, we refer to target objects such as tables as *destination objects*, the objects on the destination object are called *passive objects*, and the object manipulated by the robot is called a *manipulated object*.

template matching can facilitate their application for action execution in arbitrary environment configurations. This consequently demonstrates that a single execution model can go a long way towards equipping an agent with a mechanism for avoiding the occurrence of external faults.

Our model, whose learning process is illustrated in Fig. 1, utilises Dearden and Burbridge's [3] data storage and optimisation model and builds upon a few other approaches, such as Zampogiannis et al.'s [4] action representation and to some extent Brandl et al.'s [5] object generalisation; the manner in which we learn and generalise actions is where we diverge from these and other related models in the literature.

To demonstrate the feasibility of our approach, we consider various scenarios involving object release actions, such that the analysis focuses on four use cases: releasing rectangular prisms on top of each other, a chips can on a potentially cluttered table, a book in a container with multiple compartments, and a bottle on a fridge door's shelf. We used a KUKA youBot for experimenting with the first use case, while a Care-O-bot 3 robot was used for the other three.

II. RELATED WORK

External faults have been considered by multiple authors in the literature, such that there are three main approaches for dealing with them: preventing faults, learning from them, and performing recovery.

Relevant approaches that are concerned with preventing faults and learning from them are described by Mösenlechner and Beetz [6] and Akhtar et al. [7]. In [6], a robot-specific simulator is used for predicting action outcomes, which can be a time-consuming process. To overcome this problem, we generate execution poses by relying on a robot-independent simulation. The method described in [7] is a learning-based approach in which the expected physical behaviour of an object is identified with the help of a simulation; this knowledge is then used for finding an object state that guarantees successful action completion. The main limitation of this work is the inability to generalise, as the found solutions are too specific for a particular environment model. Recovering from external faults once they occur is another way to deal with them, a strategy often referred to as robust action execution [8], [9] and usually preceded by execution monitoring [10]. Fault recovery, while necessary for any reliable agent, cannot supply the required knowledge for executing actions correctly; in order to obtain such knowledge, our goal is to learn execution models that minimise the probability of encountering similar faults in future executions.

When it comes to action learning, both the symbolic and geometric aspects of the problem have been considered by various authors, but the two are usually studied in isolation. Symbolic learning has been addressed from multiple perspectives: an online learning method has been developed by Wang [11]; Mourão et al. [12], Pasula et al. [13], and Zhuo and Kambhampati [14] propose hypothesis learning procedures that are able to deal with noisy training data; finally, Abdo et al. [15] and Ahmadzadeh et al. [16] approach the problem from a learning by demonstration point of view. Considering our objective, both hypothesis search and learning by demonstration are rather inflexible: the former is computationally expensive and cannot easily create new models without complete retraining, while the latter requires the presence of a teacher. As presented in [11], online learning is more appropriate from our point of view - if we consider our simulation to be the required generator of expert action executions - but one of the main assumptions of [11] - noise-free data - will be violated if our simulation is not a perfect model of the real world. As none of these related symbolic learning approaches are fully applicable to our problem, we use an optimisation-based symbolic learning algorithm that we describe later on in this paper.

Symbolic models cannot be used for actual action execution, so various geometric reasoning approaches have been developed as well, such as Jiang et al.'s [17] learning-based geometric reasoning algorithm, Stulp et al.'s [18] place-dependent predictive success distribution for manipulation actions, and Dearden and Burbridge's [3] learning and reasoning approach. The algorithm in [17], which can even be used for placing unseen objects on unseen surfaces, does have attractive characteristics, but the learned model can be rather inflexible if we consider it from a relearning perspective. Unlike [17], the strategy used in [3], which couples symbolic and geometric models, is directly aligned with our objectives; that is the reason why our data encoding and pose generation strategies are based on it. The only problem with this approach is that generalisation over different environment configurations is difficult - which is something we discovered while trying to use the model as described there. Finally, the work in [18] can be seen as complementary to ours and it may be possible to extend our model using the insights presented there.[4]

III. ACTION REPRESENTATION AND LEARNING

Our work has two main objectives: (i) allowing agents to use the occurrence of external faults as a trigger for obtaining action execution knowledge autonomously and (ii) representing the obtained knowledge in a general manner so that it can be utilised for action execution in scenarios that are different from the one in which a fault has been observed. We aim to achieve these objectives by developing a representation based on which execution models are seen as task description templates that we call δ models.

A δ model, which is based on the idea that external faults are object- and task-dependent, is a combined symbolic and geometric representation of action execution knowledge that is acquired by manipulating a specific object in a set of constrained environment configurations. Both the symbolic and the geometric parts of the model - illustrated in Fig. 1 and described in sections III-B and III-C respectively - are important for generalisation beyond the scenarios on which the model was created: the symbolic component is an abstract description of the relationship between the objects in the learning configurations and is used for template matching in

[4]A more comprehensive analysis of the literature can be found in [19].

arbitrary scenarios; on the other hand, the geometric segment directs the search for execution poses to regions where execution success is more likely. The process of learning δ models, which is explained in section III-D and illustrated in Fig. 1, commences only after an external fault has been detected by an execution monitoring component and uses simulated action execution data, namely a set of poses at which the action can or cannot be executed successfully.

As δ models are strictly execution models, they are not meant to be used for high-level planning. In particular, the fact that the models are both object- and task-dependent means that they are partially grounded by definition and can only be used once a high-level plan - or a segment of a full plan - has been generated. To illustrate this point, let us suppose that a generated plan in the bottle scenario ends up with an action $putOn(Object, Surface)$. Once the variables in $putOn$ have been grounded, namely $Object$ becomes the chips can and $Surface$ the table on which the can should be placed, we can search through a list of δ models for one that corresponds to this object category and problem instance.

A. Notation

Before describing how our models are created and learned, we need to define a few terms and conventions. We represent an *object* O as a matrix that specifies the vertices of the object's axis-aligned bounding box, namely $O \in \mathbb{R}^{8\times 3}$. A *configuration* is then defined as a block matrix that combines passive objects together; for instance, a configuration with two passive objects O_1 and O_2 can be written as $C = [O_1; O_2]$, which is a 16×3 matrix. We are going to dedicate considerable attention to configurations with one and two passive objects, so we shall call those *unary* and *binary configurations* respectively. The arity of the predicates used for describing symbolic relations will be denoted by $/n$. Whenever indices are used, the notation $m : n$ means taking all indices from m to n including the limits.

B. Symbolic δ Models

The symbolic representation of δ models is based on two ideas: (i) in a given configuration of passive objects, an agent can find a place where an action could be executed by decomposing the configuration into multiple regions and examining each of those separately and (ii) the symbolic relationship between the passive objects in a well-defined[5] template configuration can be identified in a wide variety of object configurations. The first of these ideas follows from [3] and [4], while the latter influences the question of template generalisation and is the main focus of this section.[6]

Let us consider an ordered set P[7] of N predicates ($p_i \in P$ for $1 \leq i \leq N$) in which the first k predicates are unary and the last $N - k$ are binary; for instance, P might be the set $\{onTable/1, leftOf/2, rightOf/2, behind/2, inFrontOf/2\}$. With the help of P, we can convert a general binary configuration C to a symbolic representation by defining an operator s

$$s(C) = (p_{1:k}(O_1), p_{k+1:N}(O_1, O_2), p_{1:k}(O_2)) \qquad (1)$$

whose result represents a conjunction of predicates[8]. Considering our bottle scenario, if we assume that C is the configuration in Fig. 2a, the P mentioned before will result in the symbolic representation $(1, 1, 0, 0, 0, 1)$, which can be interpreted as

$$onTable(O_1) \wedge leftOf(O_1, O_2) \wedge \neg rightOf(O_1, O_2) \wedge \neg behind(O_1, O_2) \wedge \neg inFrontOf(O_1, O_2) \wedge onTable(O_2)$$

where O_1 and O_2 are the two bottles. Using s, we define a symbolic δ model as $s(C^\delta)$, where C^δ is a well-defined binary configuration. As a shorthand, we will denote symbolic δ models by S^δ.

C^δ is a *well-defined* template configuration if S^δ can be identified as an instance of a general configuration C with M objects. To search for instances of C^δ in a given configuration C, we decompose C into multiple binary configurations made of pairs of passive objects. If C contains at least one instance of C^δ, we say that S^δ *covers* C.

The concept of coverage is illustrated by the example in Fig. 2. Provided that P is a set such as $\{leftOf/2, rightOf/2, behind/2, inFrontOf/2\}$ and C is rotated by an angle of roughly $-\frac{\pi}{6}$, S^δ covers C because C's symbolic form will be equal to S^δ.

(a) C^δ (b) C

Fig. 2: A δ model constructed in C^δ covers C

As can be noticed, the definition of coverage cannot be applied to configurations that have less than two passive objects, but we can circumvent this problem by using the boundaries of the destination object as additional infinitely thin passive objects. We therefore assume that M, the number of objects in a configuration, includes both the passive objects and the boundaries of the destination object.

We can now consider a more complex example of coverage. Let us suppose that the configuration in Fig. 2a is taken to be C^δ and that our agent needs to release a can in the configuration in the upper right of Fig. 1, which is going to be our C. If we carefully observe C, we can see that multiple binary configurations, such as the configuration made of the two books and the one made of the bottle and the right table edge, are symbolically equivalent to C^δ. We can therefore

[5]The meaning of *well-defined* will become clear later in this section.

[6]A more detailed treatment of the concepts introduced in this section can be found in [19].

[7]For the purposes of this paper, we assume that P is predefined.

[8]This representation is similar to that used in [12].

say that S^δ covers C and that the action can be executed with a template execution model learned in C^δ.

By analysing the definition of coverage, we can note that an arbitrary C^δ and an unconstrained P cannot be guaranteed to produce an S^δ that will be able to cover arbitrary configurations. We can understand why by considering Fig. 3, which shows what could be C^δ and a binary configuration that will not be covered by S^δ.

(a) C^δ (b) C'

Fig. 3: An unconstrained δ model does not cover arbitrary configurations

The S^δ derived from the configuration in Fig. 3a does not cover C' because C^δ is simultaneously described by predicates such as $leftOf$ and $inFrontOf$, which can be considered equivalent under a rotation. We call configurations such as the one in Fig. 3a *ambiguous*. Under this definition, we can note that C' is not ambiguous, so taking $C^\delta = C'$ produces an S^δ that now covers the configuration in Fig. 3a as well, provided that we rotate the configuration by an angle of $-\frac{\pi}{4}$. δ models that are constructed in non-ambiguous configurations will be called *proper*. Proper δ models are the ones we are striving for when creating action execution models, as they can be generalised more easily.

The δ models described thus far can only be used for modelling actions such as placing objects on surfaces, in which the manipulated object may be subject to unwanted interactions with other passive objects. Other actions, such as putting a spoon in a cup, cannot be described with the help of multiple passive objects, as the manipulated object interacts with only one other object. Such actions will be described by $\delta^\varnothing$ models.

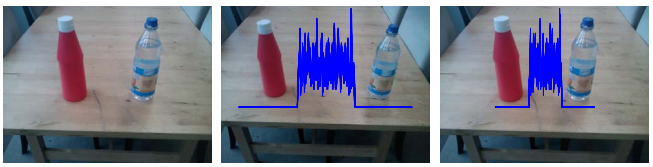
We define a symbolic $\delta^\varnothing$ model as $S^{\delta^\varnothing} = s(C^{\delta^\varnothing})$, where $C^{\delta^\varnothing}$ is a unary configuration. The symbolic form of a unary configuration depends on whether there are unary predicates in P; if the set has no unary predicates, $S^{\delta^\varnothing}$ will be empty, thus covering any arbitrary unary configuration.

C. Geometric δ Models

Symbolic δ models can only be used for verifying that an execution model is applicable to a particular configuration; for actual execution, they have to be supplemented with a geometric model that identifies feasible execution poses. For that purpose, we use (i) a *geometric sampling model (GSM)*, which is a one-dimensional histogram distribution that assigns success probabilities to points along a given dimension and (ii) Dearden and Burbridge's [3] pose optimisation algorithm that processes the samples from the GSM. For the rest of this section, we are going to concentrate on actions that can be executed on a flat surface, as that will simplify the description of GSMs.[9]

The purpose of GSMs is encoding success probabilities around the place where an action can be executed, such that any samples generated from them should direct the search for execution poses to areas where the execution is more likely to succeed. Just as symbolic δ models, GSMs are part of a template execution model, which means that they are derived from a template configuration; however, due to their geometric nature, GSMs may potentially overfit C^δ and $C^{\delta^\varnothing}$. We can understand why if we consider a book as an example of a manipulated object: if we want to place the book between two other books, using a C^δ with a large distance between the passive books will lead to an underconstrained problem in which the manipulated book can be assigned almost any orientation during execution; similarly, a C^δ in which the distance between the passive books is very small gives an overconstrained problem in which very few poses are allowed. In both cases, the obtained geometric model will not be generalisable to arbitrary configurations.

To get around this problem, we define C^δ to be a configuration in which the objects are rather close to each other - where the meaning of "close" depends on the size of the manipulated object - such that we are first going to collect data describing how to execute a given action in C^δ. The data for creating a GSM will then be collected in a configuration C^{GSM}, which has the same symbolic form as C^δ, but in which the distance between the objects is larger. The actual GSM will then be a distribution encoding the data collected in C^{GSM}, but whose boundaries are compressed so that they match C^δ. This principle is illustrated in Fig. 4.



(a) C^δ (b) C^{GSM} (c) Rescaled GSM

Fig. 4: Creating a geometric sampling model

For scaling a GSM from C^{GSM} to C^δ, we calculate a ratio between the object distances in the two configurations:

$$\gamma^\delta = \frac{x^{GSM}_{max} - x^{GSM}_{min}}{x^\delta_{max} - x^\delta_{min}} \tag{2}$$

where we assume that x_{max} is the one-dimensional position of the object whose coordinate value is larger. We then try to preserve γ^δ for each histogram bin boundary x

$$\gamma^\delta = \frac{x^{GSM} - x^{GSM}_{min}}{x^\delta - x^\delta_{min}} \tag{3}$$

such that x is calculated with respect to the object whose coordinate along a given dimension is lower. The unknown x^δ can be easily derived by rearranging equation 3.

[9]But it should be noted that GSMs are not limited to such actions.

As GSMs are histograms, they are created with a predefined resolution parameter η, which affects both the quality and the memory requirements of the representation: a lower η leads to a more fine-grained GSM, but increases the memory requirements, while a higher η reduces the memory requirements, but leads to a coarser model. A trade-off between the quality and the memory requirements is thus inevitable when choosing a value for this parameter.

The benefit of encoding GSMs in a discretised form is twofold: (i) we can easily perform importance sampling for directing the search of execution poses to areas where successful executions are more likely and (ii) they can be rescaled based on the ratio between the object distances in C^{δ} and an arbitrary configuration. Both of these properties enable generalisation to configurations that are geometrically different from C^{δ} - either because of the distance between the passive objects or because of the objects themselves. In other words, a GSM can be used for sampling positions in an arbitrary configuration if we simply rescale the histogram using equations 2 and 3, taking the histogram bin boundary in the numerator of equation 3 as the unknown variable.

As a result of this process, we obtain a model containing semantic object-specific information in addition to success probabilities. Considering the bottle example again, the data collected in C^{δ} might encode that the can should be placed between the passive bottles in a vertical orientation - which is the semantically desired one - and the GSM might assign higher probabilities to areas far from the other objects, as that is where the can is likely to avoid collisions with them.

Given that a GSM is a one-dimensional distribution, it can only be used for encoding success probabilities for a single coordinate. For an execution scenario on a flat surface, we need to create two separate GSM distributions for x and y. These two are kept separate because the distances between the objects in the two directions might be different, which means that different ratios will have to be used for rescaling the GSMs. Under the assumption that the distributions are independent, we can get a joint success distribution by multiplying the distributions for x and y; such a joint distribution is shown in Fig. 1 for the bottle scenario.

Thus far, we have only discussed GSMs in the case of δ models. For $\delta^{\varnothing}$ models, creating GSMs is much easier because we do not need a C^{GSM}; instead, GSMs are created within and around the bounding box of the only passive object in $C^{\delta^{\varnothing}}$, such that each bin boundary is calculated with respect to the minimum value of the bounding box along a given dimension. When considering a new configuration with an object whose size might be different, GSMs are rescaled with a conversion ratio that depends on the bounding box sizes of the objects in $C^{\delta^{\varnothing}}$ and the new configuration:

$$\gamma_x = \frac{max\left(O^x\right) - min\left(O^x\right)}{max\left(O^{x^{\delta}}\right) - min\left(O^{x^{\delta}}\right)} \tag{4}$$

The boundaries of the GSM's histogram bins are then shifted by preserving γ_x

$$\gamma_x = \frac{x' - min\left(O^x\right)}{x^{\delta} - min\left(O^{x^{\delta}}\right)} \tag{5}$$

such that the value of the unknown x' can be easily calculated by rearranging equation 5.

Regardless of whether we have a δ or a $\delta^{\varnothing}$ model, the search for an action execution pose is guided by the rescaled GSMs, such that a value along a given dimension is chosen by importance sampling from the respective GSM.

In addition to GSMs, geometric δ models have a second component: the geometric predicate mapping defined in [3], which is created using the geometric data collected in C^{δ} or $C^{\delta^{\varnothing}}$ for the predicates that have the manipulated object in their ground form. We use the mapping for both storing the manipulated object's orientation data and optimising the positions sampled from the GSMs. It should be noted that the GSM samples are often at a local maximum, so an actual optimisation step is not always necessary.

D. Learning and Applying δ Models

Learning a δ model is a four-step procedure: (i) we simulate C^{δ} or $C^{\delta^{\varnothing}}$ and collect a set of poses describing robot-independent action executions; (ii) a symbolic execution model is learned; (iii) a geometric mapping of the symbolic data is created; and (iv) if dealing with an action that is described by a $\delta^{\varnothing}$ model, we create GSMs from the data collected in step (i); otherwise, GSMs are created using additional data collected from C^{GSM}. The first and last steps of this process are simple if (i) we can identify whether an action should be described by a δ or a $\delta^{\varnothing}$ model, which we could do by using a knowledge base that feeds information to our model, and (ii) using the definition of proper δ models, we define the form of C^{δ} and C^{GSM}. The other two steps need a more detailed explanation.

The symbolic learning objective is to minimise a cost function c that counts the number of bit errors made by a candidate symbolic description $\bar{P}$ on a training set D^{+}

$$c(\bar{P}) = \sum_{i=1}^{R}\sum_{j=1}^{L}|\bar{P}_j - D^{+}_{ij}| \tag{6}$$

where L is the cardinality of $\bar{P}$ and R is the number of training examples. Here, D^{+} is a binary matrix in which the entries of the i-th row are derived from a set of object relations that hold before the i-th execution, namely

$$\begin{aligned} D^{+}_i = &\left(P_{1:k}\left(O_{M_i}\right), P_{k+1:N}\left(O_{M_i}, O_{1_i}\right),\right. \\ &\left. P_{k+1:N}\left(O_{M_i}, O_{2_i}\right)\right) \cup s(C^{\delta}) \end{aligned} \tag{7}$$

in the case of δ models and

$$D^{+}_i = \left(P_{1:k}(O_{M_i}), P_{k+1:N}(O_{M_i}, O_{1_i})\right) \cup s(C^{\delta^{\varnothing}}) \tag{8}$$

for $\delta^{\varnothing}$ models, where O_M represents the manipulated object and O_1 and O_2 are passive objects. $\bar{P}$ is therefore a set of ones and zeros as well, such that a value of one denotes that the predicate to which it corresponds should hold before execution. It should be noted that D^{+} is derived from the set of successful action executions in the simulated version of C^{δ} or $C^{\delta^{\varnothing}}$, which means that we do not use the negative data for learning symbolic models. In that respect, our

learning algorithm resembles the procedures commonly used in learning by demonstration [15], [16].

Instead of directly minimising c, we learn a symbolic model by maximising the fitness function

$$f(\bar{P}) = 1 - \frac{c(\bar{P})}{c_{\#}} \tag{9}$$

using hill climbing, where $c_{\#} = R \cdot L$ is the total number of errors that $\bar{P}$ can make on the training set.

Once $\bar{P}$ has been extracted, we lift the predicates and split them into two segments: the predicates whose ground form includes the manipulated object and have a value of one are used for creating a geometric predicate mapping, while the predicates that are instantiated with only passive objects are memorised as a representation of the template model. As mentioned before, a geometric predicate mapping is created using the kernel density-based representation defined in [3]; both the positive and the negative data are used here.

Using a δ or a $\delta^{\varnothing}$ model for executing an action is a four-step procedure as well: (i) the objects in a scene are recognised and used for instantiating a robot-independent simulation - which works best if the objects are known; (ii) an instance of the action's symbolic model is identified in the configuration by only considering pairs of objects in the case of δ models and single objects in the case of $\delta^{\varnothing}$ models; (iii) a set of robot-independent successful execution poses is collected by simulating the execution at numerous poses, all of which are sampled from a rescaled version of the model's GSMs and are subsequently optimised just as in [3]; and (iv) a robot can select one of these poses for execution after sorting the set according to certain criteria,[10] such as reachability or distance from its current pose.

In principle, having a geometric predicate mapping created with data from a template configuration means that we may need to extrapolate over the geometric data in the optimisation process of the third step. To avoid this problem, the simulated passive objects can be shifted once an initial pose has been sampled so that the distance between them corresponds to that in the training set; the optimisation can then proceed as if working in the template configuration.

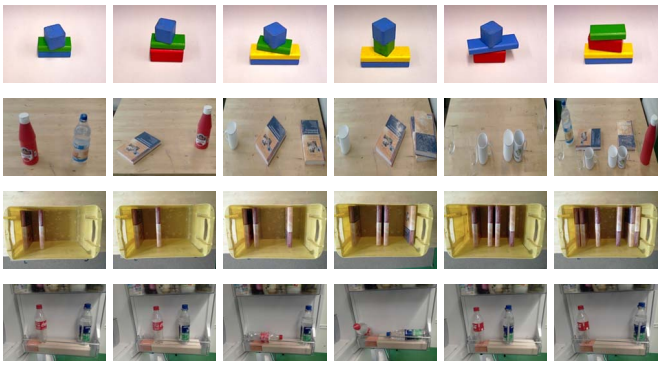
IV. EXPERIMENTS

To demonstrate how δ models can be used for acquiring execution knowledge, we consider object release actions in the context of four use cases: stacking blocks, placing a chips can on a table, releasing a book in a horizontal container with multiple compartments, and putting a bottle on a fridge door's shelf. The objective in the first use case is to build a tower of three blocks, such that the block on the bottom is placed manually and a robot has to put the other two blocks on top. In the other use cases, the objective is to release the manipulated object on top of or in the destination object and keep the poses of the passive objects close to their initial values, namely within $2.5cm$ of the position and 5^o of the orientation. In the table and fridge use cases, the manipulated object should stand upright after execution; in the container use case, the book can have any arbitrary final orientation.

For collecting training data and generating robot-independent execution poses, we used a custom-made Unreal Engine[11] simulation. We experimented with the block tower use case using a KUKA youBot, while a Care-O-bot 3 robot was used for the other three use cases.[12]

A. Experimental Setup

We designed six scenarios for each use case, shown in Fig. 5. We assume that a fault occurs in scenario A of each use case; the fault triggers the learning process and the learned model is then used for execution in all six scenarios.



(a) A (b) B (c) C (d) D (e) E (f) F

Fig. 5: Use cases from top to bottom: block tower, table, container, fridge

The execution model for the block tower use case is a $\delta^{\varnothing}$ model that specifies how to put a cube on top of another cube. Scenario A of this use case only demonstrates that the execution model can be easily generalised to other blocks, so the tower has only two blocks; a tower of three blocks is built in scenarios B-F. In the table, fridge, and container use cases, scenario A represents a binary configuration that was used for creating the respective δ models, so scenarios B-F have the goal of demonstrating that the models can be generalised to configurations different from C^{δ}.

For training the models, we collected 5000[13] randomly generated execution poses around the regions where the actions could be successfully executed. A symbolic model was then learned using the positive training examples, namely 52 training examples in the block tower use case, 3830 examples in the table use case, 3481 examples in the fridge use case, and 571 examples in the container use case. For symbolic learning, we used a predicate set P that combines predicates from Zampogiannis et al. [4] and Akhtar et al. [7]:

$$\begin{aligned} P = \{&on/2, leftOf/2, rightOf/2, above/2, \\ &below/2, behind/2, inFrontOf/2, ntpp/2, \\ &ntpp_i/2, ec/2, dc/2, eq/2, po/2\} \end{aligned}$$

[10] It should be noted that we do not deal with the criteria used for selecting poses as they may be application-specific.

[11] https://www.unrealengine.com

[12] A video showing some of the experiments can be found at https://youtu.be/Q8vSwQT-bf8

[13] A large number of training examples is actually not essential for these use cases, but may be useful to have in general.

The last six predicates in P are part of the region connection calculus (RCC8) [20], such that we merged two RCC8 predicates - tpp and tpp_i - with $ntpp$ and $ntpp_i$ respectively.

Just as described in [3], both the positive and the negative data were used for creating a geometric predicate mapping. The GSMs in the table, fridge, and container use cases were created with additionally collected 1000 training examples. In all four use cases, we used a resolution parameter $\eta = 0.1$.

Once the execution models were learned, a robot was used for releasing an object 10 times in each scenario; we therefore had 240 experimental trials in total. As our study is not focused on grasping, we were manually putting the manipulated object in the robot's gripper before each trial. For the table, container, and fridge use cases, we collected a set of 60 robot-independent successful execution poses before each set of 10 trials, which means that there were six candidate poses for each execution. We assumed that the robot could move itself to a position close to each of the poses, so the six candidates were ordered based on whether a collision with the passive objects was expected - the opening radius of the gripper was used for verifying that a collision will not occur - and whether the release height could be reached by the manipulator. For the block tower use case, we only collected 10 robot-independent successful execution poses for each set of 10 trials; neither reachability nor collisions had to be considered here because the blocks we used were small and the manipulated block was always released above the other blocks.

B. Results

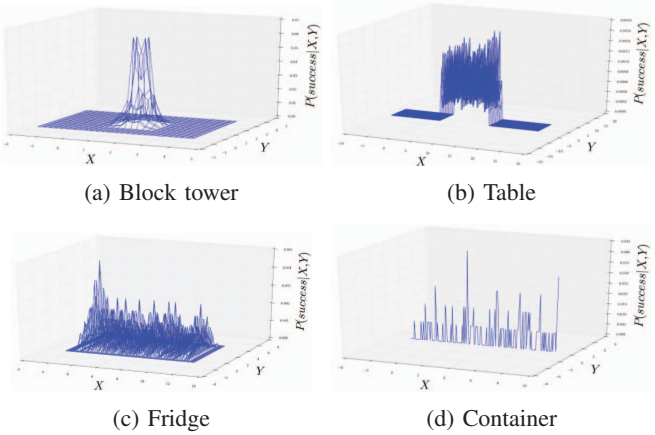
The learned symbolic models of the four use cases can be interpreted as in Table I, where the symbolic templates are shown in boldface.

TABLE I: Learned symbolic models

Fridge	$rightOf(O_M, O_1) \wedge dc(O_M, O_1) \wedge$ $leftOf(O_M, O_2) \wedge dc(O_M, O_2) \wedge$ $\mathbf{leftOf(O_1, O_2)} \wedge \mathbf{dc(O_1, O_2)}$
Table	$rightOf(O_M, O_1) \wedge dc(O_M, O_1) \wedge$ $leftOf(O_M, O_2) \wedge dc(O_M, O_2) \wedge$ $\mathbf{leftOf(O_1, O_2)} \wedge \mathbf{dc(O_1, O_2)}$
Container	$rightOf(O_M, O_1) \wedge above(O_M, O_1) \wedge$ $dc(O_M, O_1) \wedge leftOf(O_M, O_2) \wedge above(O_M, O_2) \wedge$ $dc(O_M, O_2) \wedge \mathbf{leftOf(O_1, O_2)} \wedge \mathbf{dc(O_1, O_2)}$
Block tower	$above(O_M, O_1) \wedge po(O_M, O_1)$

By observing the models, we can note that the fridge and table use cases have the same symbolic representation, which is not surprising considering the similarity of the execution scenarios with respect to the given P. It can also be noted that the S^δs of the fridge, table, and container use cases are the same, encoding that O_1 should be to the left of O_2. As there are no unary predicates in P, the template model of the block tower use case is empty.

Fig. 6 shows the joint GSM distributions for the four use cases. As can be seen, the GSMs for the fridge, table, and block tower use cases encode precisely what we would expect: in the first two cases, the expected success drops near the passive bottles (and the edge of the shelf for the fridge); in the last case, success can only be expected within the boundaries of the passive block. We can also note that the container use case has no GSM along y; this is because the compartments in C^δ have no y separation whatsoever, so the joint GSM is a one-dimensional distribution.



(a) Block tower (b) Table

(c) Fridge (d) Container

Fig. 6: Use case GSMs

Finally, table II shows the number of successful executions for each use case and scenario.

TABLE II: Successful runs (out of 10)

	A	B	C	D	E	F
Fridge	9	10	9	10	8	8
Table	10	10	10	10	10	8
Container	10	10	10	10	10	10
Block tower	9	8	8	8	8	10

As we can see here, the success rate is consistently high for all use cases and scenarios, though a few failed executions can be noticed as well. In some of the more difficult scenarios, the manipulators themselves could not reach the goal poses accurately enough. The fact that the number of training examples was relatively low - particularly in the block tower use case - is another possible reason for these failures, as certain regions of the execution space might not have been adequately covered by the training data. The robot-independent simulation might have contributed to these failures as well, but we need more experiments for verifying the validity of that hypothesis.

In principle, each of these failures could have been used as an opportunity for relearning the execution models; however, the main objective of our analysis was to demonstrate that actions modelled by δ and $\delta^\varnothing$ models can be executed with a high success rate even in configurations that are significantly different from the ones used for creating the models, which is why we learned only one model per use case.

V. DISCUSSION AND CONCLUSIONS

The action execution model presented in this paper is based on various assumptions, all of which affect the applicability of our work to general robotic agents. We assume that the modules necessary for executing actions, such as perception, navigation, grasping, and trajectory planning, are all relatively reliable; this is the reason why aspects related to

them are not included in the model representation. Another major assumption is that an agent starts with an initial set of execution rules, which means that the action effects are known; they are in fact used for labelling the simulated training sets. [14] We also assume that the objects encountered by a robot are known and that there are more or less accurate 3D models of them. [15]

The manner in which δ models are defined also introduces certain limitations to our model. First of all, deciding whether an action should be described by a δ or a $\delta^{\varnothing}$ model is not easy; an ontology that defines object properties may simplify this problem, but designing such an ontology can be complicated as well. Another potential problem is that it might be difficult to generalise the template models to configurations with passive objects that differ from those in C^{δ}; in other words, knowing how to manipulate a given object in the presence of one category of objects does not necessarily mean that the object will have the same behaviour in the presence of other objects, so deciding how to create a template model can be a daunting task. From a practical point of view, it should also be mentioned that our geometric pose generation procedure does not take the rotations of the passive objects into account; in the case of δ models, the rotation suggested by the geometric model can be easily corrected if a configuration has to be rotated for a symbolic match to be found, but the course of action is not so clear if the configuration does not need a full rotation, but the individual objects are rotated nonetheless. It is also possible to raise an objection about the fact that δ models are both object- and instance-dependent, which seemingly makes them unsuitable for generalisation to anything other than a very constrained set of scenarios. This is only true to a certain extent, as the models are not meant to exist in isolation and can benefit from knowledge bases that categorise objects in semantic classes or affordance-based methods such as [22].

Despite their limitations, we believe that δ models have modelling power and can be useful for avoiding the occurrence of external faults that are caused by physical phenomena. The results presented in section IV support this conclusion, although a more detailed analysis with a larger variety of use cases might demonstrate this more conclusively. There are a few aspects that future work could address to improve our work; in particular, it would be desirable to improve the formalisation of template configurations, define a strategy that uses semantically related model representations for biasing the model learning process, and expand the framework so that the effects of actions can be learned as well.

ACKNOWLEDGMENTS

We would like to thank Iman Awaad for her useful insights and Alex Moriarty for comments that improved our manuscript.

[14] While it may seem unreasonable to know the effects before having the preconditions, we can easily imagine a scenario in which a robot is told what to do - the effects - but not how to do it - the preconditions.

[15] There are ways to avoid the severe consequences of this assumption, for instance by utilising a knowledge base such as the one described in [21].

REFERENCES

[1] R. Isermann, *Fault-Diagnosis Systems: An Introduction from Fault Detection to Fault Tolerance*. Springer, 2006.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[3] R. Dearden and C. Burbridge, "Manipulation Planning using Learned Symbolic State Abstractions," *Robotics and Autonomous Systems*, vol. 62, no. 3, pp. 355–365, 2014.

[4] K. Zampogiannis, Y. Yang, C. Fermller, and Y. Aloimonos, "Learning the Spatial Semantics of Manipulation Actions through Preposition Grounding," in *Robotics and Automation (ICRA), 2015 IEEE Int. Conf.*, May 2015, pp. 1389–1396.

[5] S. Brandl, O. Kroemer, and J. Peters, "Generalizing Pouring Actions Between Objects using Warped Parameters," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS Int. Conf.*, Nov. 2014, pp. 616–621.

[6] L. Mösenlechner and M. Beetz, "Parameterizing Actions to have the Appropriate Effects," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2011, pp. 4141–4147.

[7] N. Akhtar, A. Kuestenmacher, P. G. Plöger, and G. Lakemeyer, "Simulation-based approach for avoiding external faults," in *Int. Conf. Advanced Robotics (ICAR)*, 2013.

[8] A. Steck and C. Schlegel, "SmartTCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots," in *Proc. Int. Workshop Dynamic languages for Robotic and Sensors systems (DYROS/SIMPAR)*, November 2010, pp. 274–277.

[9] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[10] O. Pettersson, "Execution monitoring in robotics: A survey," *Robotics and Autonomous Systems*, vol. 53, pp. 73–88, 2005.

[11] X. Wang, "Learning Planning Operators by Observation and Practice," in *Proc. 2nd Int. Conf. Artificial Intelligence Planning Systems*, 1994, pp. 335–341.

[12] K. Mourão, L. S. Zettlemoyer, R. Petrick, and M. Steedman, "Learning STRIPS Operators from Noisy and Incomplete Observations," in *Proc. 28th Conf. Uncertainty Artificial Intelligence*, 2012, pp. 614–623.

[13] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning Symbolic Models of Stochastic Domains," *Journal Artificial Intelligence Research*, vol. 29, no. 1, pp. 309–352, May 2007.

[14] H. H. Zhuo and S. Kambhampati, "Action-model Acquisition from Noisy Plan Traces," in *Proc. 23rd Int. Joint Conf. Artificial Intelligence*, ser. IJCAI '13, 2013, pp. 2444–2450.

[15] N. Abdo, H. Kretzschmar, L. Spinello, and C. Stachniss, "Learning Manipulation Actions from a Few Demonstrations," in *Robotics and Automation (ICRA), 2013 IEEE Int. Conf.*, May 2013, pp. 1268–1275.

[16] S. R. Ahmadzadeh, A. Paikan, F. Mastrogiovanni, L. Natale, P. Kormushev, and D. G. Caldwell, "Learning Symbolic Representations of Actions from Human Demonstrations," in *Robotics and Automation (ICRA), 2015 IEEE Int. Conf.*, May 2015, pp. 3801–3808.

[17] Y. Jiang, C. Zheng, M. Lim, and A. Saxena, "Learning to Place New Objects," in *Robotics and Automation (ICRA), 2012 IEEE Int. Conf.*, May 2012, pp. 3088–3095.

[18] F. Stulp, A. Fedrizzi, and M. Beetz, "Action-Related Place-Based Mobile Manipulation," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ Int. Conf.*, Oct. 2009, pp. 3115–3120.

[19] A. Mitrevski, "Improving the Manipulation Skills of Service Robots by Refining Action Representations," Master's thesis, Bonn-Rhein-Sieg University of Applied Science, January 2016.

[20] D. A. Randell, Z. Cui, and A. G. Cohn, "A Spatial Logic based on Regions and Connection," in *Proc. 3rd Int. Conf. Knowledge Representation and Reasoning*, 1992, pp. 165–176.

[21] M. Tenorth, A. C. Perzylo, R. Lafrenz, and M. Beetz, "The RoboEarth language: Representing and Exchanging Knowledge about Actions, Objects, and Environments," in *Robotics and Automation (ICRA), 2012 IEEE Int. Conf.*, May 2012, pp. 1284–1289.

[22] M. Schoeler and F. Wörgötter, "Bootstrapping the Semantics of Tools: Affordance Analysis of Real World Objects on a Per-part Basis," *IEEE Transactions on Cognitive and Developmental Systems*, vol. 8, no. 2, pp. 84–98, June 2016.