Simulation-Based Debugging of Formal Environment Models*

Tim Meywerk[1], Arthur Niedzwiecki[2], Vladimir Herdt[1,3] and Rolf Drechsler[1,3]

Abstract— **Logic-based formal models of robot environments are often used to aid the generation and verification of robotic plans. They are however often simplified and rather abstract compared to the real world that the robot acts in. This can lead to considerable discrepancies between the behavior of the formal model and that of physics-based simulation engines. These discrepancies are not always apparent to the designer. In this paper we propose a new methodology to make these discrepancies explicit by combining formal verification and simulation. Our approach is able to find relevant discrepancies, while only requiring a small number of simulations.**

I. Introduction

Robots working in complex environments like households face various challenges such as a dynamic environment and a variety of different tasks. A promising approach to handle this complexity is the usage of generalized robotic plans. During the creation of such plans, logical formalisms like the *event calculus* [1, 2] or *situation calculus* [3, 4] are often employed to model the robot's environment. More recently, formal models have also been used to verify the correctness of existing plans [5]. Formal models allow for exhaustive reasoning, but the rigid framework of these formalisms often means that formal models are simplified and rather abstract compared to the real world that the robot acts in.

There are two main reasons for the higher abstraction in formal models. One is the complexity of real-life physics, that can often not be adequately modeled in terms of formal logic. Another reason is the discretization that often takes place, i. e. the environment is partitioned into a finite set of discrete positions instead of using real-valued coordinates. Depending on the level of abstraction, this can lead to considerable discrepancies between the behavior of the formal model and that of physics-based simulation engines. However, when used in planning and verification, formal models are usually assumed to be correct. Discrepancies in the model can have severe consequences. A plan derived or verified from a faulty model is often also faulty and can result in considerable damages to the robot and its environment. Unfortunately, discrepancies of formal models are not always apparent to the designer and to the best of our knowledge, there is no systematic approach to find them.

In this paper we aim to make these discrepancies explicit by combining formal verification techniques with robotic simulation. The main idea is to use the existing formal verification engine SEECER [5, 6], which is based on the *Discrete Event Calculus* (DEC) [7], to find particularly interesting execution traces in the formal model and then run the same execution in the simulator. The resulting states of both executions are then compared. Our approach is able to focus on specific robotic plans and specific interesting final states. This way we need to perform only very few simulation runs compared to a naive brute force approach.

Physics-based simulation is a standard tool to produce safe and high-quality robotic plans. Simulation engines like Gazebo [8] or Webots [9] come with a variety of features and accurate physics simulation. Simulation-based testing of robotic plans is the de-facto standard way to assess their correctness and find errors.

Formal verification [5, 10] on the other hand tries to formally prove the correctness of plans. Since a robot interacts with its environment, it is usually not sufficient to verify the plan by itself. Instead, formal models of the robot's environment are developed and used in reasoning. These models are often logic-based, for instance employing the situation or event calculus.

Many plan-based robotic systems also employ runtime verification [11], which aims to monitor the robots actions during runtime. In contrast to formal verification it can only find errors while they occur and can not guarantee the absence of errors in any way.

The combination of formal verification and simulation is a promising direction for the scalable verification of complex systems. One such technique is concolic testing [12], which combines symbolic execution of software programs with concrete random test cases. Thus far, hybrid formal verification and simulation methods have usually been used to verify a software or hardware system. The application to verification of formal models has to the best of our knowledge not yet been attempted.

The debugging of formal models is still a manual process. Similarly to the software domain, tools that assist in debugging have been developed [13]. However, they still require an experienced developer to identify errors in the formal model. Our approach aims towards a mostly automatic debugging process, where only the modifications to the formal model have to be done manually.

The remainder of this paper is structured as follows: In Section II we provide the background necessary for the understanding of this paper. Section III explains our approach to finding discrepancies between model and simulation in detail. In Section IV we evaluate our approach in three practical scenarios, followed by a discussion on the limitations and applications of our approach in Section V. Section VI concludes the paper.

*The research reported in this paper has been supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 *EASE – Everyday Activity Science and Engineering*, University of Bremen (`http://www.ease-crc.org/`). The research was conducted in sub-project P04.

[1]Research Group of Computer Architecture, University of Bremen, Germany
[2]Institute for Artificial Intelligence, University of Bremen, Germany
[3]Cyber Physical Systems, DFKI GmbH, Bremen, Germany
{tmeywerk, aniedz, vherdt, drechsler}@uni-bremen.de

```
1  (let ((?goal-pose *origin-pose*))
2    (perform
3      (a motion
4        (type going)
5        (pose ?goal-pose))))
6  (let ((?looking-pose *looking-pose*))
7    (perform
8      (a motion
9        (type looking)
10       (pose ?looking-pose))))
11 (perform
12   (a motion
13     (type detecting)
14     (object (an object
15         (type ?object-type))))
```

Fig. 1: Excerpt of a CPL plan for object detection

II. Preliminaries

In this section, we present the preliminaries necessary for the understanding of our approach. These include the Cognitive Robot Abstract Machine in Section II-A as well as the verification engine *SEECER*, which is used to formally verify the correctness of robotic plans in Section II-B.

A. Cognitive Robot Abstract Machine

The *Cognitive Robot Abstract Machine* (CRAM) [14] is a powerful toolbox for the generation, execution and simulation of robotic plans. It is able to automatically infer action parameters depending on the situation at hand. It also monitors the task execution in real-time and has several ways to detect failures and recover from them.

A core module of CRAM is the *CRAM Planning Language* (CPL), which is used to formulate general robotic plans. It is built upon the *Common Lisp* programming language and is therefore a Turing-complete language. Figure 1 shows an excerpt of a CPL plan. It consists of three `perform` statements in Lines 2, 7 and 11. Each `perform` statement has as argument a motion *designator*. A designator is a possibly underspecified description of an action that the robot should execute. The first motion designator in Line 3 instructs the robot to navigate to a predefined goal position. The exact path or speed is not specified within the plan, but is instead inferred at runtime by a motion planner. Similarly, the second designator in Line 8 makes the robot turn its head towards a specified position and the third designator in Line 12 uses a perception subroutine to search for objects of a certain type within the robot's field of view.

Another important module within CRAM is the fast projection simulator [15] based on the Bullet physics engine. Due to some simplifications in the physics calculation it is able to achieve a very fast simulation speed. This allows CRAM to simulate the effects of several possible action parametrizations before the best parametrization is executed on the real robot. Despite its slight simplifications, the simulator has been shown to accurately predict an action's effects when performed on a real robot.

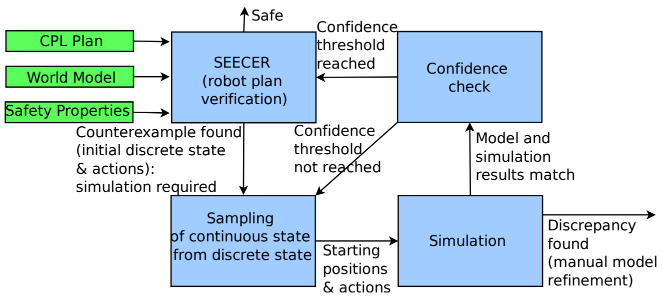
CPL Plan
World Model
Safety Properties
Safe
SEECER (robot plan verification)
Confidence threshold reached
Confidence check
Counterexample found (initial discrete state & actions): simulation required
Confidence threshold not reached
Model and simulation results match
Sampling of continuous state from discrete state
Starting positions & actions
Simulation
Discrepancy found (manual model refinement)

Fig. 2: Overview of our approach

B. Formal verification for CRAM

Formal verification for plan-based robotics is still in its infancy. One approach specifically tailored to CRAM is the verification engine SEECER [5, 6]. SEECER combines symbolic execution [16] of the CPL source code with environment modeling and reasoning based on the *Discrete Event Calculus* (DEC) [7]. Symbolic execution is a well established technique from software verification. It works by executing all possible paths through the program and using symbolic variables instead of concrete values for all inputs. During execution, constraints over those variables are collected and solved by an SMT solver. This way, a symbolic execution engine can identify inputs that reach certain states or violate certain properties. The DEC is a logical formalism based on many-sorted first-order logic. It used the sort of *fluents* and *events* to describe a dynamic system, actions that can be executed in that system and those actions' preconditions and effects.

SEECER is given a CPL plan, an environment description and a set of *safety properties*. These safety properties describe certain constraints of the robots environment that should always hold. For instance, the robot should never place an object at a position that is already occupied by another object. SEECER can now decide whether there is an initial state and a viable path through the plan code, such that the resulting final state of the environment violates one or more safety properties.

Usually safety properties are used to describe certain properties that need to hold for the safety of the robot and its environment. However, in general any property that constrains a finite number of states can be specified, irrespective of whether they are related to safety or not. To reflect this slightly different semantic, we will refer to them simply as *state constraints* for the remainder of this paper.

III. Finding Discrepancies

In this section we present our approach to detect discrepancies between a formal model written in the DEC and a simulated environment. We begin with some definitions and a general overview of the approach in Section III-A. Afterwards, we describe the sampling and confidence calculation in greater detail in Sections III-B and III-C.

A. Overview

To find discrepancies, we need an initial state of the robot and its environment, such that the same chain of action executions results in different final states in the formal model and the simulation.

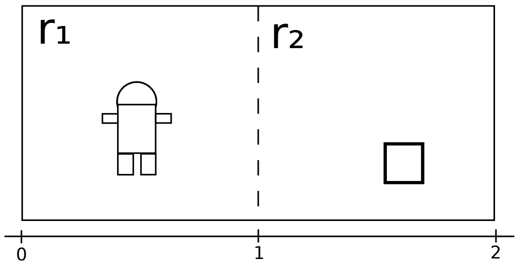

Fig. 3: Simple robotic environment

Here, a state is a mapping from parameters such as positions or angles to their values. However, states in the formal model are usually different from states in the simulation, since the simulation uses real numbers to describe the parameters, while the formal model has to be discrete. This discretization is usually implemented by modeling a finite set of discrete values for each parameter, which correspond to regions and intervals in the continuous space. In the following, we will denote a state over continuous, i. e. real numbers as a *continuous state* and a state over discrete positions and angles as a *discrete state*. To semantically connect continuous and discrete states, we introduce a mapping $m : S_c \mapsto S_d$, where S_c and S_d are the sets of continuous and discrete states, respectively. Consequently, each continuous state maps to a single discrete state, while each discrete state is mapped to by an infinite amount of continuous states. Additionally, we define an *execution trace* as a sequence $s_0, a_0, \ldots, a_{n-1}, s_n$ of states s_i and actions a_i. Action a_i is executed in state s_i and results in a new state s_{i+1}. We call s_0 the *initial state* and s_n the *final state*. Whenever it is not clear from context, we will use the terms *continuous execution trace* and *discrete execution trace* to denote whether the s_i are continuous or discrete states.

Example 1. Consider the simple environment depicted in Figure 3. It consists of a rectangular area, which is divided into two regions r_1 and r_2. The regions are defined through their x-coordinates with r_1 spanning from $x = 0$ to $x = 1$ and r_2 spanning from $x = 1$ to $x = 2$. The environment contains a robot (currently in r_1) and a box that the robot can pick up and move (currently in r_2). A plan might now instruct the robot to move to $x = 1.5$, pick up the box, move back to $x = 0.5$ and place it there. This would produce a discrete final state with both the robot and the box inside r_1. The continuous state would be more accurate and contain the exact positions of robot and box.

Figure 2 shows the high-level flow of our approach. The input to the procedure consists of three components (green boxes in Figure 2): a plan written in CPL, a world model and a set of state constraints. The world model and state constraints are formalized as a DEC description. The state constraints describe a set of discrete states which should trigger a simulation using the same fluents and predicates as the world model. These three inputs are fed into our symbolic verification engine SEECER (top left in Figure 2), which will then try to find a discrete execution trace which leads to one of the desired states. We modified SEECER in such a way that it does not terminate after the first finding, but instead reports the execution trace and waits for the rest of the procedure to finish before the symbolic execution is continued. Whenever SEECER returns an execution trace, the initial state $s_{d,0}$ and the action sequence are extracted. To set the initial state in the simulator, it has to be converted into a continuous state. This is done through sampling (bottom left), i.e. selecting a state $s_{c,0}$ with $m(s_{c,0}) = s_{d,0}$ at random. Afterwards, the initial state $s_{c,0}$ and the actions are given to the simulator (bottom right), which sets the initial state, executes the actions and then compares the resulting final state against the one found by SEECER. If the final continuous state from the simulation does not map to SEECER's final discrete state, a discrepancy between the model and simulation has been found. This discrepancy is then reported and the procedure terminates. If the simulator and the DEC model reach matching final states, there is no discrepancy for the sampled initial state. However there may be another initial continuous state mapping to the initial discrete state which causes a discrepancy. Due to the infinite amount of continuous states, exhaustive sampling is not possible. Instead we use a stochastic approach and calculate the confidence in the hypothesis that there is no problematic continuous state mapping to $s_{d,0}$ (top right). If this confidence reaches a pre-defined threshold, the execution trace is assumed to not cause any discrepancy and SEECER continues to search for the next execution trace. Otherwise, a new initial state is sampled and simulated. The following subsections provide more information on the sampling process and the confidence calculation.

B. Sampling-Based Simulation of Counterexamples

Due to the difference between discrete and continuous states, there is no unique continuous state for each discrete state returned by SEECER. Instead, a continuous state has to be sampled. The sampling process and the subsequent confidence calculation is easiest, if all parameters are sampled independently of each other. This means that the discrete world model describes rectangular or cuboid regions. We will focus on this case in the following discussion. Once all parameters have been sampled, the initial state is set in the simulation. Afterwards the action sequence is executed. The resulting final state can now be compared with the final discrete state returned by SEECER.

Example 2. Consider again the environment and plan from Example 1. Sampling an initial concrete state might result in the robot at $x = 0.8$ and the box at $x = 1.4$. After setting this state in the simulation and executing the actions, both the robot and box would be at $x = 0.8$. This final continuous state maps to the final discrete state with both the robot and the box in r_1.

The sampling and simulation is repeated until either a discrepancy has been found or a pre-defined confidence is reached. We describe the confidence calculation in the following subsection.

C. Calculating the Confidence

In this section, we describe the calculation of the confidence that occurs after each simulation run. To do that, we consider the action sequence as a function, which maps every initial continuous state to the respective final continuous state. We assume that this function follows the multivariate normal distribution, i. e. each parameter of the final state is a linear combination of the parameters of the initial state plus a normally distributed error ϵ. Each parameter y of the final state is given as $y = \sum_{i=1}^{n}(x_i\beta_i) + \epsilon$, where n is the number of parameters, x_i is the i-th parameter value of the initial state and β_i is its coefficient. This equation can also be written in matrix notation as $y = X\beta + \epsilon$, where X is the row vector $\begin{pmatrix} x_1 & \dots & x_n \end{pmatrix}$ and β is the column vector

$$\begin{pmatrix} \beta_1 \\ \dots \\ \beta_n \end{pmatrix}.$$

As usual, we also assume that all parameters of the final state are independent of each other given a fixed initial state.

Since the coefficients β are unknown, they have to be estimated from the sampled initial states and their respective final states. We call these estimated coefficients b. Using the sampling data, the equation can now be rewritten as $y = Xb + \epsilon$, where y is now a column vector of the sampled parameter from the final state. X is a matrix, where each row corresponds to a sample and each column corresponds to a parameter from the initial state. ϵ is now a column vector as well, containing the error for each sample.

Example 3. Consider again the simple environment from the previous examples. The state can be fully described through the x-coordinate of the robot and the box. Lets assume, we are interested in the final position of the box and we have two samples: In the first sample the robot and box start at $x = 0.5$ and $x = 1.3$, respectively, and the box ends up at $x = 0.6$ in the final state. In the second sample the robot and box start at $x = 0.9$ and $x = 1.7$ and the box ends up at $x = 0.8$. The equation would now be written as

$$\begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix} = \begin{pmatrix} 0.5 & 1.3 \\ 0.9 & 1.7 \end{pmatrix} b + \epsilon$$

Following the standard least squares procedure, we want to minimize the sum of the squares of those errors, i. e. the term $\sum_{i=1}^{n} {\epsilon_i}^2 = \epsilon'\epsilon$. The respective values of b can be estimated by $b = (X'X)^{-1}X'y$ and the least squared error by $\sigma := min(\epsilon'\epsilon) = (y - Xb)'(y - Xb)$. Once all coefficients and errors for all parameters of the final state have been estimated, we can calculate a confidence interval for each of them. This confidence interval is given by $[\sum_{i=1}^{n}(x_{i,min} \cdot b_i) - \frac{Z(\sqrt[n]{C})\sigma}{\sqrt{s}}, \sum_{i=1}^{n}(x_{i,max} \cdot b_i) + \frac{Z(\sqrt[n]{C})\sigma}{\sqrt{s}}]$, where $x_{i,min}$ is the smallest possible value of x_i if $b_i > 0$ and the highest possible value of x_i otherwise. Similarly, $x_{i,max}$ is the highest possible value if $b_i > 0$ and the smallest possible value otherwise. Z is the Z-distribution, C the desired confidence, n the number of parameters and s the number of samples. Using $\sqrt[n]{C}$ makes sure, that the confidence of the combination of all n confidence intervals is $\sqrt[n]{C}^n = C$. This joint confidence region now describes the possible final states that we expect to reach from any initial continuous state mapping to $s_{d,0}$. If now all final states in the confidence region map to the same discrete state, we can terminate the sampling loop and continue with the next execution trace.

Example 4. Consider the environment from the previous examples once again. Assume, that the coefficients

$$b = \begin{pmatrix} 0.9 \\ -0.1 \end{pmatrix}$$

and the least squared error $\sigma = 0.3$ have been found after sampling 4 initial states. Further assume, that the initial discrete state has the robot in region r_1 and the box in r_2. Since r_1 is bound by $0 \leq x \leq 1$ and the coefficient $b_1 = 0.5$ is positive, we use $x_{1,min} = 0$ and $x_{1,max} = 1$. On the other hand, $b_2 = -0.4$ is negative, so we use $x_{2,min} = 2$ and $x_{2,max} = 1$. Using a 95% confidence, we get $Z(\sqrt{0.95}) \approx 2.24$. Plugging these values into the above equation, results in a lower interval boundary of

$$0 \cdot 0.9 + 2 \cdot (-0.1) - \frac{2.24 \cdot 0.3}{2} = -0.536$$

and an upper boundary of

$$1 \cdot 0.9 + 1 \cdot (-0.1) + \frac{2.24 \cdot 0.3}{2} = 1.136$$

Since both interval boundaries are outside of the region r_1, the desired confidence is not yet reached and more simulations need to be performed.

The following section presents our experimental evaluation.

IV. Experimental Evaluation

In this section, we evaluate our proposed approach on three robotic plans set in a household kitchen environment.

The underlying formal model, written as a DEC description, is identical for all three plans except for the types of items that are present in the kitchen. The formal model uses the DEC axioms and consists of an additional 7 sorts, 2 predicates, 16 fluents, 10 events and 59 logical sentences. For the most part, it describes pre-conditions and effects of the actions used in the plans, such as navigation, pick-up or drawer access. On the simulation side, we use the fast plan projection simulator [15], that is tightly integrated with CRAM. For the formal verification we use our extension of SEECER.

All experiments were conducted on a Linux machine running an Intel Core i5-7200U CPU with 2.50 GHz clock rate.

In the following Section IV-A we present the three robotic plans used in the evaluation. Section IV-B contains the experimental results and the discrepancies that have been found.

A. Robotic Plans

All three of our plans operate in the same kitchen and therefore share the same formal model and simulation environment. All plans are concerned with pick-and-place tasks, i. e. transporting items from one part of the kitchen to another. They have been selected, since they are well-suited to showcase the types of discrepancies that can be found with our approach. Below, we describe all plans in further detail.

1) Plan 1: Setting the Table: The first plan is tasked with setting a table for breakfast, i. e. transporting a bowl, a cereal box, a milk carton and a spoon from the kitchen workspace

TABLE I: Simulation data until a first discrepancy is found

plan	iterations	simulations	time
Plan 1	1	12	273s
Plan 2	2	11	197s
Plan 3	1	1	22s
Plan 2 (modified)	3	20	289s

to a nearby table. The spoon is located in one of the three available drawers, all other items are on top of the workspace.

The plan loops through the items and has the robot transport each one individually. In case of the bowl, cereal and milk, the robot attempts to detect the item on top of the workspace. For the spoon, the robot searches through the drawers by opening them, trying to detect the spoon and then closing them again. Once an object has been detected, it is picked up, the robot navigates to a pre-defined position in front of the table, and the object is placed in its target position.

2) Plan 2: Bowl and Spoon: This plan is a modified version of the first one. This time only the bowl and the spoon are transported to the table, the spoon is transported first and the drawers are only closed as long as the spoon has not been detected.

3) Plan 3: Looking into Drawers: This plan uses the spoon inside one of the drawers again and no other items. The goal again is to find the spoon inside the drawers and then transport it to the table. However, the spoon is now allowed to be not only in the center of the drawers, but also towards the side or very far in the front or back. Therefore, the robot has to try to detect the spoon from multiple poses per drawer. These poses are all close together and are therefore inside the same region described by the formal model.

B. Experimental Results

We executed the proposed approach on the three robotic plans using a confidence threshold of 99%. We were able to find three major discrepancies between the formal model and the simulator.

The first discrepancy was found during execution of Plans 1 and 2. The state constraints were chosen to trigger a simulation after a successful navigation action, i. e. whenever the robot executed a navigation action to some position p at timepoint t and actually reached position p at timepoint $t+1$. Both in simulation and when executing on a real robot, it may happen, that the robot loses an object from its gripper, either due to a bad grasp or sudden movement. This object will then usually fall to the floor or a surface, often outside of the robots vision or reach. In the formal model this case was not considered. The main effect of a navigation action, namely the new position of the robot, was formalized, but no changes to other fluents such as gripper attachment or objects' positions were made.

The second discrepancy also occurred after a successful navigation action during Plan 2. Part of the plan is a loop that opens a drawer, searches for the spoon inside and then closes the drawer. However, when the spoon was actually found, the closing was omitted, i.e. one drawer would always stay open. Depending on the robots position, this drawer would sometimes block the path that the robot was supposed to take. Thus, the success of the navigation action would sometimes depend on the state of the drawer. This was correctly reflected inside the simulation, since collision checks are done before each navigation action. If there is no free path between the origin and goal positions, the action would fail. The formal model did not contain any such constraint, though. This is a typical oversight by the model engineer, which can be easily fixed. In addition to the error in the formal model, this discrepancy also uncovered a flaw in the robotic plan, which can now be modified to always close the drawer, even if the spoon was found.

The final discrepancy occurred in Plan 3. This time, a successful grasping action was used as the state constraint. Plan 3 uses several positions to search for the spoon inside a drawer. Since all of those positions are relatively close together compared to the total size of the kitchen, they fall into the same region. The formal model assumes that all perception and grasping actions from this region succeed if the object is inside on of the drawers and that drawer is currently open and unobstructed. In practice however, the small differences in positions were crucial in the visibility and reachability of the spoon. The discretization used in the formal model was simply too coarse to accurately reflect the true conditions for visibility and reachability of the spoon. A finer discretization would be able to mitigate this problem, but this would of course also increase the size of the model and the complexity of reasoning.

The detected discrepancies clearly show that our approach can effectively find discrepancies in formal models. In addition, our experimental results indicate that our approach can work very effectively in keeping the number of required simulation runs low for finding these discrepancies. To show this aspect, we recorded experimental data during the execution. They are summarized in Table I. The first column *Plan* states the plan that was executed. The next column shows the number of iterations, i. e. how many execution traces were returned by SEECER and then used for sampling. The third column reports the number of simulation runs and the final column contains the total time spent in the simulation.

Here, we have the three plans described above, as well as a modified version of Plan 2, where both the plan and the formal model now consider the discrepancies found previously. Consequently, no discrepancy was found for this modified version. For Plan 1 and Plan 3 the first execution trace led directly to a discrepancy, while Plan 2 first produced an execution trace, where no discrepancy was found. Instead, after 7 samples the confidence threshold of 99% was reached. The discrepancy later found in the second execution trace could in fact not occur in this first execution. The modified Plan 2 needed 3 iterations until all execution traces were explored. In all cases, only very few runs were necessary to find the discrepancies. This also led to a small amount of time spent in the simulation. In all cases, less then 5 minutes of simulation time were used.

This is evidence that our approach is able to effectively find relevant discrepancies, while only requiring a small number of simulations.

V. Discussion and Future Work

The evaluation showed the practical applicability of our approach. This section discusses how its results should be interpreted and how the approach can be tuned to fit the user's needs. We also give several directions for future work.

The approach is correct, since every discrepancy reported by the algorithm necessarily has to be observed to actually produce two different final states. It is however not complete. This is due to the infinite number of concrete states for any discretization. While the absence of a discrepancy can not be formally proven, our approach can give a probabilistic guarantee by employing the measure of confidence. The confidence threshold can be freely chosen by the user. They can easily increase the chance to find even very rare or hidden discrepancies by increasing the threshold. This will of course also increase the number of simulation runs and therefore the runtime.

The results of our approach can be used in multiple ways. The obvious choice is to refine the formal model in such a way that the discrepancy no longer occurs. However, sometimes such a refinement may make the model vastly more complicated and thus make reasoning harder. Alternatively, the discrepancies could be collected and any result derived from the formal model could be reviewed with regard to the discrepancies. This could be done either manually or (semi-)automatically through simulation. So far, it was always assumed that a discrepancy means that the formal model is faulty. There may of course also be the case where the formal model is accurate, but the simulation engine is not. While we expect this case to occur rarely in practice, the discrepancy could as well be used to modify the simulation engine.

Right now, our approach regards the simulator as a pure black box. In future work, we also want to leverage some knowledge about the simulator's internal functionality. This could allow to get a higher degree of certainty in the case that no discrepancy could be found, maybe even up to a full formal proof of the equivalence of model and simulation. We also want to investigate the case where the state's parameters are not fully independent. This way, not only rectangular and cuboid regions could be considered, but also other shapes.

One could also use our proposed approach to build a formal model from scratch. A developer would start with some kind of minimal model, e. g. a model where all actions have no effect. Afterwards our approach is used to detect any discrepancies between the model and a simulator. These discrepancies are then used to manually refine the model. This process is repeated until no further discrepancies are found. We expect a model produced in this way to be very well adapted to the plan(s) and state constraints used. On the other hand, it should be minimal in a sense, i. e. contain no sentences that are irrelevant to the plan(s). This in turn should lead to high realism of the model while keeping the reasoning effort low.

VI. Conclusion

In plan-based robotics, formal models are a widely used tool to assist in creating and verifying plans. These formal models allow for exact and exhaustive reasoning, but are usually more abstract than simulation. The discrepancies in behavior between formal models and simulation are rarely obvious. In this paper we proposed a novel methodology that makes these discrepancies explicit for the first time.

Our approach combines formal verification and simulation and can be targeted towards specific robotic plans and environment states. The main loop first uses the formal verification tool SEECER to find interesting execution traces and extract the initial state and the action sequence. From the initial discrete state a continuous state is sampled. This is then fed into the simulation engine. If the resulting final state of both executions do not match, a discrepancy has been found. Otherwise the sampling is repeated until a sufficient confidence is reached.

Our experimental evaluation clearly shows that the approach is able to find interesting discrepancies in real-world scenarios. Furthermore, it did so with very few simulation runs.

References

[1] R. Kowalski and M. Sergot, "A logic-based calculus of events," in *New Generation Computing*, vol. 4, 1986, pp. 67–95.

[2] R. Miller and M. Shanahan, "Some alternative formulations of the event calculus," in *Computational Logic: Logic Programming and Beyond. Lecture Notes in Computer Science*, vol. 2408, 2002, pp. 452–490.

[3] J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in *Machine Intelligence 4*, 1969, pp. 463–502.

[4] H. Levesque, F. Pirri, and R. Reiter, "Foundations for the situation calculus," 1998.

[5] T. Meywerk, M. Walter, V. Herdt, J. Kleinekathöfer, D. Große, and R. Drechsler, "Verifying safety properties of robotic plans operating in real-world environments via logic-based environment modeling," in *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2020, pp. 326–347.

[6] T. Meywerk, M. Walter, V. Herdt, D. Große, and R. Drechsler, "Towards Formal Verification of Plans for Cognition-enabled Autonomous Robotic Agents," in *Euromicro Conference on Digital System Design (DSD)*, 2019, pp. 129–136.

[7] E. T. Mueller, "Event Calculus Reasoning Through Satisfiability," in *Journal of Logic and Computation*, 2004, pp. 703–730.

[8] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004, pp. 2149–2154.

[9] O. Michel, "Cyberbotics ltd. webots™: Professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, 2004.

[10] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," in *ACM Computing Surveys*, 2019, pp. 1–41.

[11] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Roşu, "Rosrv: Runtime verification for robots," in *International Conference on Runtime Verification*, 2014, pp. 247–254.

[12] K. Sen, "Concolic testing," in *22nd International Conference on Automated Software Engineering (ASE)*, 2007, pp. 571–572.

[13] R. Mannadiar and H. Vangheluwe, "Debugging in domain-specific modelling," in *Software Language Engineering*, 2011, pp. 276–285.

[14] M. Beetz, L. Mösenlechner, and M. Tenorth, "Cram—a cognitive robot abstract machine for everyday manipulation in human environments," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010, pp. 1012–1017.

[15] L. Mösenlechner and M. Beetz, "Fast temporal projection using accurate physics-based geometric reasoning," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 1821–1827.

[16] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, 2018.