Towards Formal Verification of Plans for Cognition-enabled Autonomous Robotic Agents*

Tim Meywerk[1] Marcel Walter[1] Vladimir Herdt[1] Daniel Große[1,2] Rolf Drechsler[1,2]

[1]Group of Computer Architecture, University of Bremen, Germany
[2]Cyber Physical Systems, DFKI GmbH, Bremen, Germany
{tmeywerk, m_walter, vherdt, grosse, drechsle}@informatik.uni-bremen.de

***Abstract*—In this paper, we propose the first approach for verifying plans of cognition-enabled autonomous robots that perform everyday manipulation activities in human environments. Our methodology is based on the new Intermediate Plan Verification Language (IPVL) which is used to represent plans, environments, and robot belief states in one joint formal model. We devise a symbolic execution engine for IPVL and show the effectiveness of our overall verification methodology in a case study.**

***Index Terms*—Cognitive robotics, Robot control, Formal verification**

I. Introduction

For decades, robots have only been used for repetitive tasks in fixed environments, and their main application was to make high-volume production in manufacturing facilities cheaper. Advances in technology, artificial intelligence and engineering allowed to build autonomous robots, which are able to perform tasks in unpredictable environments, to learn, and to adjust their behavior.

Meanwhile, the social acceptance of robotic systems has increased, and robots have found entrance into the household. A prominent example are robotic vacuum cleaners. However, enabling a robot to perform complex everyday manipulation activities, like e. g. setting the table or preparing a meal autonomously, poses several challenges to the development of the robot control system. Essentially, robots need to be equipped with cognitive mechanisms, which allow them to deduce what kind of action is suitable to achieve a desired task goal. This includes, but is not limited to, applying different grasp types for different objects and positioning themselves spatially to be able to reach out to a location. As a consequence, the control programs of cognition-enabled autonomous robots use high-level behavior specification languages, which allow to infer control decisions instead of requiring pre-programmed decisions. Several specialized high-level behavior specification languages have been developed in the past. Examples are RPL [1], RMPL [2], and CPL [3]. They all share certain attributes like their inherent concurrency and the ability to call perception, navigation, and manipulation tasks.

While non-trivial scenarios of everyday manipulation activities can be mastered today, the complexity of the plans is steadily increasing. At the same time, simulation of these plans and even testing on physical robots reach computational limits. This causes concern about the safety of autonomous service robots; especially those interacting with humans or handling potentially dangerous items. Hence, formal verification techniques are necessary to ensure the safety of involved humans and the robots themselves.

Related Work General task planning is a wide research field. Originally suggested by [4], robot motion planning has been enriched with requirements logic [5] in the last decade. As one example, the temporal logic *LTL* has been used to express search control knowledge in planning [6], [7]. In the context of autonomous robotic agents and planing of non-deterministic actions, several formalisms like e. g. *Hierarchical Task Networks* [8], [9], *Markov Decision Processes* [10], or *Situation Calculus* [11] have been developed. On top, logic languages have been defined (see e. g. *GOLOG* [12]). For these robot control programs, also verification of temporal properties has been investigated [13], [14]. For large state spaces, [15] proposed coverage-driven verification to validate robotic code nevertheless. However, no formal verification approach targeting cognition-enabled robotic plans has been proposed so far.

Contribution In this paper, we propose the first approach for verifying plans of cognition-enabled autonomous robots that perform everyday manipulation activities in human environments. We use the toolbox *Cognitive Robot Abstract Machine* (CRAM) [3] for realizing the cognition-enabled robot control program. In particular, CRAM provides the *CRAM Plan Language* (CPL), which captures high-level plans in Common Lisp. For the CPL, we envision a verification methodology based on *Symbolic Execution* (SE) [16], [17] as it has been shown that SE is a highly effective technique for finding deep errors in complex software applications. The foundation of our approach is the new *Intermediate Plan Verification Language* (IPVL) which serves as a formal intermediate representation. Our approach compiles CPL plans down to IPVL and integrates environment models as well as robot belief states into a single IPVL description. We additionally devised the *Symbolic Execution Engine for Cognition-Enabled Robotics* (SEECER), which is tailored for IPVL. SEECER allows to check plan correctness with respect to environment models as well as annotated assumptions and assertions.

*This work was supported by the German Research Foundation (DFG) as part of the Collaborative Research Center (Sonderforschungsbereich) 1320 *EASE – Everyday Activity Science and Engineering*, University of Bremen (http://www.ease-crc.org/) in subproject P04, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

To keep this paper self contained, the following Section II recapitulates and defines all concepts used in this work. Section III presents the main contribution i. e. the approach for verifying CPL plans. We give a case study in Section IV and conclude with Section V.

II. Preliminaries

In this section, we give definitions and concepts necessary for understanding the rest of this paper. At first, we introduce Common Lisp, followed by the plan language CPL. Afterwards, we give a short overview of symbolic execution and introduce the Wumpus World, which will be used as a running example and case study.

A. Common Lisp

The ecosystem around the Lisp programming language is a de facto standard in many robotics, AI, and general decision making systems. A lot of code is already available supporting or written in Common Lisp (a dialect of Lisp). The CPL introduced in the next section is an extension of Common Lisp.

To keep this paper self-contained, we give a quick overview on Common Lisp's code structure. All of Common Lisp's build-in symbols, basic units, and numbers are called *atoms*. A *list* is a sequence of either atoms or other lists separated by blanks and surrounded by parentheses. With that in mind, we can define *s-expressions* (short for *symbolic expressions*) in a recursive manner: an atom is an s-expression. If $s_1, \ldots, s_n$ are s-expressions, then the list $(s_1 \ldots s_n)$ is also an s-expression. If an s-expression is intended to be evaluated, it is called a *form*. Such an evaluation takes the s-expression's first element s_1 as the function name and all the other elements $s_2, \ldots, s_n$ as its arguments. A Common Lisp program is a sequence of forms.

Example 1. *Consider the code snippet depicted in Listing 1. It consists of 9 non-atomic s-expressions, which are nested and perform simple arithmetic under a condition. Fixed values are assigned to the variables* `a` *and* `b` *by the* `let*` *keyword, which is used to define variables in a local scope.*

B. CRAM Plan Language

For this work, we consider control programs for the autonomous robots written as high-level plans in the *CRAM Plan Language* (CPL) [3]. Plans describe desired behavior in terms of hierarchies of goals, rather than fixed sequences of actions that need to be performed. An architectural overview of the CRAM ecosystem, to which CPL belongs, is depicted in Fig. 1. A CPL plan receives additional information from the robots belief state and knowledge base via a query-answer architecture. It also activates *Perception* and *Manipulation & Navigation* modules, which then get information from or act on the *Environment*. These interaction calls happen in the form of *designators*.

Designators are a common concept employed in several reasoning and planning systems. They are often implemented as data types encapsulating high-level descriptions of entities

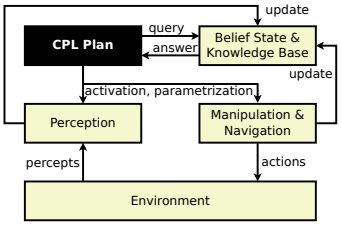


Fig. 1: Overview of the CRAM stack architecture

```
(let* ((a 10) (b 12))
  (if (> a 0)
    (* 3 (+ a b))
    (* 3 b)))
```

Listing 1: Common Lisp

```
(defun place-object (?target-pose ?arm)
  (par
    (perform (a motion (type looking)
      (target (a location (pose ?target-pose)))))
    (perform (an action (type placing)
      (arm ?arm) (target (a location (pose ?target-
    pose)))))))
```

Listing 2: CPL High-Level Plan

familiar to humans, but abstract to robots. Classes of designators available in CPL are for instance

- *location designators*: physical locations under constraints like reachability, visibility, etc.,
- *object designators*: real world objects on a semantic level like what they are and what they could be used for,
- *human designators*: description of a human entity within an environment, and
- *motion and action designators*: actions that can be performed by a robotic agent.

In CPL, an action designator contains the action type to perform (like perceiving or grabbing) and several parameters. It can be passed to the `perform` function, which breaks it down to sub-tasks and takes care of their execution. Both action designators and the `perform` function are particularly important for this work and will be investigated further in Section III-B. The following example illustrates a typical use of different designators.

Example 2. *Listing 2 shows a typical high-level CPL plan using multiple designators. Designators are generated using the* `a` *keyword. The plan in Listing 2 performs a motion to turn the robot's head to look at a specified target position and places the robot's arm to the same location in parallel. As can be seen, designators may be nested, such as the two location designators used by the action and motion designator.*

Next a brief introduction to symbolic execution is given.

C. Symbolic Execution

Symbolic Execution (SE) analyzes the behavior of a program pathwise by treating inputs as symbolic values. The (symbolic) program state is represented by a set of symbolic expressions, which are assigned to the program variables, and a (Boolean) path condition pc, which must be satisfied by the instructions. Initially, pc is set to `true`, i. e. there are no constraints on the symbolic expressions. Along an execution path s, the program state is updated according to the execution semantic of each

instruction. An assignment instruction overwrites the value of a variable with the right-hand-side expression. At each branch instruction, the execution path s is *split* into two independent paths $s_{\mathtt{t}}$ and $s_{\mathtt{f}}$ due to two possible evaluations of the branch condition c. The pc for each path is updated accordingly as $pc(s_{\mathtt{t}}) := pc(s) \wedge c$ and $pc(s_{\mathtt{f}}) := pc(s) \wedge \neg c$, respectively. Only feasible paths will be explored further. A path is feasible iff its pc is satisfiable. For verification purposes, *assume(c)* adds c to the current pc to prune irrelevant paths (i. e. a path is pruned if the new $pc := pc \wedge c$ is not satisfiable) and *assert(c)* checks for assertion violations, i. e. $pc \wedge \neg c$ is satisfiable. Satisfiability checks are performed by an SMT solver. A more comprehensive overview on SE can be found in [18].

D. The Wumpus World

Autonomous robotic agents find themselves in highly complex environments, which exceed the limits of exhaustive reasoning. It is incredibly hard to operate within these environments; even simulated behavior pushes computing systems to their limits.

The *Wumpus World* [19] is an environment that operates on relatively simple rules, but still poses a challenge due to its incomplete information available to agents.

Definition 1. *The Wumpus World is defined as a rectangular grid of cells to which cartesian positive integer coordinates are assigned. We define* $(0, 0)$ *to be the left-most bottom position. Position values increase in northern and eastern direction respectively.*

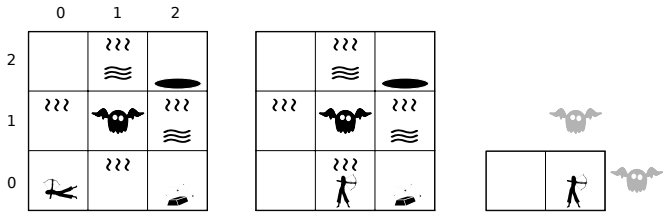
The Wumpus World is assumed to be a dungeon where every cell represents a room. An agent can enter and leave the Wumpus World in room $(0, 0)$ *only. In every room, doors to adjacent ones can be found. Though, the agent's perception is limited to events in its current room.*

The agent's goal is to find a glittery nugget of gold placed in one of the rooms, pick it up, and leave the dungeon safely. Attempting doing so, the agent might face obstacles in terms of the Wumpus*, a dangerous creature emitting a bad odor to adjacent rooms, and a number of deep pits, around which in adjacent rooms a light breeze can be perceived. Facing either the Wumpus or a pit, the agent will be eaten alive by the Wumpus or fall to death respectively. Neither the Wumpus nor the pits change their positions.*

For its defense, the agent is equipped with a single arrow, which can be shot in any orthogonal direction at any time within the dungeon. Arrows cross rooms until they hit a wall or the Wumpus. The latter leads to the Wumpus' death and the immediate ending of bad smells in adjacent rooms.

To interact with the world, the agent may perform several actions: turning*,* walking*,* grabbing*,* shooting*,* climbing*, and* perceiving*.*

Example 3. *Consider the* 3×3 *Wumpus World in Fig. 2a. An agent equipped with bow and arrow just entered the dungeon and is located in position* $(0, 0)$*. A glittery gold nugget is placed at position* $(2, 0)$*, whereas there is a Wumpus in room* $(1, 1)$ *and one pit at position* $(2, 2)$*.*



(a) A 3×3 Wumpus World (b) Environment after one walking action (c) Agent's belief state

Fig. 2: Environment vs. belief state

Odors emitted by the Wumpus are depicted as vertical curvy lines while breezes swirling around pits are horizontal ones. The glittery shine of gold is represented as twinkling stars.

How the environment looks in reality and how it is perceived by the agent, can differ. When initially proposed, the Wumpus World was designed to be a simple to understand concept, but within its rules providing only incomplete information to agents.

Example 4. *Assume, the agent in the Wumpus World depicted in Fig. 2a would turn to its right and would walk to room* $(1, 0)$*. The resulting world is shown in Fig. 2b. A perceiving action for bad odors would tell the agent that a Wumpus is close. Though, they do not know where it is precisely. Moreover, they do not even know about the size of the world, they found themselves in.*

Fig. 2c depicts the agent's belief state after the first walk and perception. They know, they started somewhere and walked in eastern direction. Since there, they detected a bad smell, a Wumpus might hide in either adjacent room.[1]

In the next section, we present our approach for verifying plans of cognition-enabled robots.

III. Formal Verification of CPL Plans

In this section, we propose a verification approach for plans of cognition-enabled autonomous robotic agents based on symbolic execution. We start with an overview and the general idea. We then go into detail about how we deal with plan-environment-interaction via an interface. Afterwards, we present our own *intermediate representation* (IR) for plan verification and finally close with a detailed description of symbolic execution on that very representation.

A. Overview

This section summarizes the approach proposed in this paper before we go into more detail in the following sections. Consider Fig. 3 for an overview. Our goal is to formally verify that certain safety constraints on a given CPL plan hold.

We start with compiling the CPL plan into our own IR. For that, we use a language, which we call IPVL and which we describe in Section III-C.

[1] In some implementations, the agent does not know on which position they start, leading to a belief state suspecting a Wumpus even in southern direction. In case they know they are in $(0, 0)$ in the beginning – like our agent does – the concern for a Wumpus in southern direction is obviously eliminated.

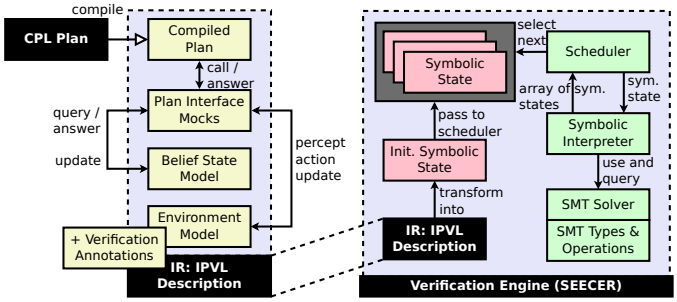


Fig. 3: Overview of proposed plan verification approach

Additionally, we integrate environment models as well as agent belief states into the IR. Integrating the environment model allows reasoning about the agent's actions. The IR plan accesses these IR models by means of *mocked* functions. Essentially, these *mocks* are models of the corresponding CPL plan interface functions. They enable the IR plan to perform perception, navigation, and manipulation tasks on the environment model and query the belief state model.[2]

For verification purposes, symbolic expressions in combination with assumptions and assertions (verification annotations) are embedded into the IR (see lower left yellow box in Fig. 3). This enables a comprehensive state space exploration.

Finally, the combined IR description is passed to a verification engine to check for assertion violations triggered by the plan execution.

Our contribution includes (1) the IPVL to act as an IR, and (2) SEECER, which is tailored for IPVL.

IPVL is compact, yet powerful enough to capture the simulation semantics of cognition enabled robotic plans in combination with the agent's belief state and environments. SEECER checks plan correctness with respect to the environment model and the specified assumptions and assertions.

In the following, we present more details on modeling plan interface functions and the environment (Section III-B) as well as our IPVL (Section III-C) and SEECER (Section III-D).

B. Plan Interface and Environment Modeling

In this section, we describe how we integrate environments into our verification process and model the interaction between environments, agents, and plans. We already discussed in Section II-B that modern robotic agents utilize planning languages to initiate interaction with environments. CPL defines the `perform` (plan interface) function, which can be called in Common Lisp. `perform` allows to initiate perception, navigation, and manipulation tasks. The following example gives an intuition of how `perform` calls work.

Example 5. *Reconsider the* 3×3 *Wumpus World given in Fig. 2a with the agent located at the left bottom corner, i. e. position* $(0, 0)$*, facing in northern direction. A glittery gold nugget is still placed at position* $(2, 0)$*. Assume further, the agent makes the following sequence of* `perform` *calls:*

```
(perform (an action (type turning) (direction right)))
(perform (an action (type walking)))
(perform (an action (type walking)))
(perform (an action (type perceive) (signal glitter)))
```

The first call would make it turn to its right (i. e. in eastern direction). The second and third call would make them walk in eastern direction (i. e. the direction they face) to position $(1, 0)$ *and* $(2, 0)$*. The final call would detect a glittery object at that very location.*

Even though `perform` offers an intuitive interface for programmers, the underlying complexity of calls like `perform` in planning languages is non-trivial in domains of symbolic execution. As Fig. 3 touches, calls to knowledge bases for instance are common. By *mocking* the `perform` function, we are able to handle the agent's initiated actions in a way that simulates the desired environment without (1) the agent being actually in it and (2) calling the whole underlying planning stack. Mocking in this context means creating a function, which to the plan *behaves* like `perform` would do without calling the underlying stack; and such reducing complexity. For discrete and finite worlds such as the Wumpus World, we use the intended behavior for every possible `perform` call to dynamically create this mock for that very function in Common Lisp yielding a complete set of rules for the desired environment.

This is a general concept that applies to any concrete environment and planning language. For our ongoing example, we utilize CPL and the Wumpus World.

In Algorithm 1, we give a pseudo code description of the `perform` mock. An input action designator d is checked for its type in Line 2. Dependent on that type, actions are performed. In the Wumpus World, these can be of type *turning* (Line 3), *walking* (Line 7), *grabbing* (Line 10), *shooting* (Line 14), *climbing* (Line 19), and *perceiving* (Line 21).

A walking action for example makes the agent take one step in its viewing direction if it is not facing a wall. If they do, a *bump* signal is triggered instead, that can be perceived by the agent to let them know, they walked into a wall. Any action besides perceiving makes the bump signal disappear again (Line 1).

Due to page limitations, we cannot go into full detail here and therefore omit the *perceiving* implementation in the pseudo code (Line 22). Implementing perceiving is straightforward as it contains another `switch` over the signal to perceive, e. g. *glitter* or *stench*, and returns `true` iff such a signal is present in the agent's current room.

Please Note that our approach allows for the modeling of both deterministic and non-deterministic environment models through the use of additional symbolic variables and assumptions.

In general, the perform mock can be expressed in any programming language or formalism and is then translated to IPVL. This way, we need only one interpreter and no designated environment model. We introduce our IPVL in the next section.

[2]Note that it is possible to exchange the environment model without modifying the plan; hence, to verify the same plan's safety in different environments.

Algorithm 1: CPL `perform` mock

```
Input: Action designator d
if type(d) ≠ perceiving then bump ← false
switch type(d) do
    case turning do
        dir ← direction(d)
        if dir = right then turn 90° clockwise
        else turn 90° counterclockwise
    case walking do
        if agent faces a wall then bump ← true
        else go one step in viewing direction
    case grabbing do
        if agent_x = gold_x ∧ agent_y = gold_y then
            remove the gold nugget from the world
            has_gold ← true
    case shooting do
        if has_arrow then
            has_arrow ← false
            if Wumpus is located in viewing direction then
                remove Wumpus from world
    case climbing do
        if agent_x = 0 ∧ agent_y = 0 then leave dungeon
    case perceiving do
        ...
```

C. Intermediate Plan Verification Language

We define the language features of IPVL in this section. We start with definitions and examples of IPVL's core and then introduce compiling Common Lisp to IPVL by linearization.

Whichever planning language might be used by a robotic system, one only needs to implement a compiler for translating it to IPVL in order to get symbolic execution and verification mechanisms with SEECER on top.

1) Core Language: We especially designed the IPVL to make a translation as easy as possible. IPVL is Turing-complete, dynamically typed (like Common Lisp and many other languages in robotics), and incorporates an Assembly-like paradigm.

IPVL code is a sequential list of instructions. Such a representation is general and at the same time much more manageable for a verification back-end (e. g. similar concepts are adopted by LLVM or CBMC). However, it requires to linearize functional languages like Common Lisp.

IPVL uses simple arithmetical, logical, comparison, and conditional instructions. In combination with variable assignments, `goto`s, function calls, and special verification instructions, the whole language can already be described.

The instruction set has been designed to be as simple as possible to allow a compact verification backend. On the other hand it should be complex enough to express plans written in higher level robotic languages like CRAM.

The IPVL acts as a interface between a verification frontend such as annotated CRAM and a verification backend like SEECER, and allows for both parts to be developed independently.

The most common instruction in IPVL is that of an assignment, where the left-hand-side is a variable name and the right-hand-side is either a constant or an expression. Expressions of arithmetical, logical, and comparison type have at most two parameters.

```
1 v := 21 * 2;
2 w := v - 4;
3 x := w / 3.8;
4 y := w > x;
5 z := y and true;
```

Listing 3: Assignments

```
1 i := 0;
2 loop:
3 i := i + 1;
4 c := i < 10;
5 if c goto loop;
```

Listing 4: Loops

```
1 defun sq ( x )
2 r := x * x;
3 return r;
4 end defun;
5 y := sq ( 10 );
```

Listing 5: Functions

```
1 x := a < 100;
2 assume x;
3 y := a * b;
4 z := y > 50;
5 assert z;
```

Listing 6: Verification

The following example illustrates IPVL's core instruction classes.

Example 6. *Listing 3 demonstrates assignment instructions. As you can see, on the right-hand-side either constants (`21`, `2`, `4`, `3.8`, `true`) or variables (`v`, `w`, `x`, `y`, `z`) are allowed.*

Listing 4 demonstrates the instantiation of a `for` loop that counts from `0` to `10` with `i` as the loop variable. The expression `loop:` defines a label called loop *at line 2. If the condition `c` in line 5 is fulfilled, execution will jump back to that label. This also allows us to construct `while` and `do while` loops.*

In Listing 5, a function declaration together with a corresponding call is depicted. The function `sq` with parameter `x` is defined by the block `defun` ... `end defun`. Functions must incorporate a `return` instruction to specify a designated return value.

Finally, Listing 6 shows our special verification annotations `assume` and `assert`. While `assume` specifies a value assumed to be true as per definition, `assert` is a request to proof that the following cannot be false.

2) Linearizing: When compiling functional languages to IPVL, certain problems arise. In the following, we give an example of how Common Lisp's `let*` function needs special handling.

Example 7. *Reconsider the snippet from Listing 1. It incorporates a `let*` function, which assigns some variables. Though, these variables are only valid within `let*`'s scope. IPVL does not incorporate a concept like scopes, leading to a possible problem when the variables continue being valid.*

Therefore, we introduced the `unlet` instruction to IPVL to get rid of previously assigned variables. Translating the snippet from Listing 1 to IPVL yields the code shown in Listing 7, that makes use of `unlet`.

We developed a decomposition mechanism to transform Common Lisp's tree structure to sequential IPVL code. Of course, more cases are to be handled than shown in Example 6 and Example 7. Though, due to page limitations, we rather give an intuition than a full specification of IPVL because this would be out of scope of this paper.

```
a := 10;
b := 12;
_temp0 := a > 0;
if _temp0 goto then;
_result := 3 * b;
goto endif;
then:
_temp1 := a + b;
_result := 3 * _temp1;
endif:
unlet a;
unlet b;
```

Listing 7: Intermediate Linearized Representation

We describe symbolic execution for IPVL in the next section.

D. Symbolic Execution for IPVL

In this section, we present our Symbolic Execution Engine SEECER for IPVL, that was mentioned over the previous sections. The right part of Fig. 3 shows an overview of SEECER's architecture. Essentially, SEECER consists of a scheduler and a symbolic interpreter. The scheduler manages a set of symbolic execution states and orchestrates the state space exploration by selecting, which state to consider next. The selected state is passed to the interpreter for symbolic execution. IPVL instructions are interpreted one after another while the symbolic execution state is updated accordingly. The interpreter returns to the scheduler in one of three cases: (1) the end of the IPVL program is reached, (2) an unsatisfiable assumption is reached, or (3) a branch instruction with symbolic condition is executed. In the third case the interpreter will split the symbolic execution state into two independent states and return these two states to the scheduler for further processing. The interpreter employs an SMT solver to check for assertion violations and check feasibility of symbolic branch instructions. Besides user specified assertions, our interpreter also checks for generic execution assertions, e. g. zero divisions.

SEECER starts with a combined IPVL description (which, as described in Section III-A, integrates the environment model, the belief state model, and the actual plan). The IPVL description is transformed into an initial symbolic execution state, which is then passed to the scheduler. The scheduler performs a *Depth First Search* (DFS). DFS is a common state space exploration strategy that focuses on each path individually and thus is memory efficient (which is important in handling large state spaces). SEECER terminates either after finding a violated assertion or after exploring the whole state space. In the latter case, the plan is shown to be correct with respect to the environment model and the specified assumptions and assertions.

In the following, we present more details on symbolic execution states and our symbolic interpreter.

1) Symbolic Execution State: A symbolic execution state can be defined as the tuple (pc, ip, α). pc is the path condition, which describes the preconditions needed to reach the current path. The instruction pointer ip points to the next IPVL instruction to be executed. The mapping α stores the current value of all variables. It maps each variable name to a *cell*.

A cell can contain any structure that may be formulated in current CPL plans. We support integer values, reals, Booleans, strings, symbols (in the Common Lisp sense), designators, functions, and lists. Integers, reals, Booleans, and strings are represented as SMT expressions and may contain both symbolic or concrete values. Symbols and designators can only contain concrete values. In a typical CPL plan, those symbols and designators will be used as constant values, which means that there is no need to support symbolic values. Functions consist of a pointer to a segment of IPVL code and a list of parameter names. The function pointer is also handled concretely, while the parameters can be symbolic expressions. Lists are assumed to have a concrete size, contain cells itself, and may be concrete, symbolic or a mix of both.

The engine starts with ip pointing to the first line of a given IPVL description. This corresponds to the entry point of the CPL plan. The path condition pc is set to `true` and the mapping α is empty.

2) Symbolic Interpreter: The interpreter executes IPVL instructions one after another and updates the symbolic execution state accordingly. Every instruction except for conditional and unconditional `goto` instructions increases the ip by one. Whenever a new variable is introduced via an assignment or the `sym` instruction in IPVL, a new cell is added to α. Symbolic variables introduced via a `sym` instruction are mapped to a new symbolic SMT variable. The `assume` and `assert` verification annotations are executed according to their usual semantic (see Section II-C). If the SMT solver detects an assertion violation, the engine will terminate and return a *counterexample* (CEX). Based on the CEX, it is possible to retrieve the state of the environment model and CPL plan as well as the assertion that has been violated.
For branch instructions "`if` c `goto` l", two cases are considered:

(1) Only one branch direction is feasible. Then, the interpreter will continue with the next instruction ($pc \wedge \neg c$ is satisfiable, but $pc \wedge c$ is not) or the instruction at label l ($pc \wedge c$ is satisfiable, but $pc \wedge \neg c$ is not). No scheduler interaction is involved in this case.

(2) Otherwise (both directions feasible). Then, the current state s is replaced with two new states $s_\mathtt{t}$ and $s_\mathtt{f}$, defined as follows:

$$\begin{aligned} pc(s_\mathtt{t}) &:= pc(s) \wedge c & pc(s_\mathtt{f}) &:= pc(s) \wedge \neg c \\ ip(s_\mathtt{t}) &:= l & ip(s_\mathtt{f}) &:= ip(s) + 1 \\ \alpha(s_\mathtt{t}) &:= \alpha(s) & \alpha(s_\mathtt{f}) &:= \alpha(s) \end{aligned}$$

Essentially, $s_\mathtt{t}$ continues as if c was `true` and $s_\mathtt{f}$ as if c was `false`. Please note, an SMT solver is only employed if c is symbolic. Furthermore, only one clone operation is necessary to obtain $s_\mathtt{t}$ and $s_\mathtt{f}$, because the current state s is re-used. The interpreter returns $s_\mathtt{t}$ and $s_\mathtt{f}$ to the scheduler.

Other instructions like arithmetic or logical ones will manipulate the cells in α according to their execution semantics. They are mapped to SMT expressions in a straightforward way.

We conducted a case study by assembling all individual components described in the previous sections. These include the approach to compile plans and environment models to

IPVL enriched with safety annotations as well as our verification engine SEECER to test those annotations. In the following section, we give an overview of our results.

IV. Case Study: The Wumpus World

We have implemented our verification approach for plans of cognition-enabled autonomous robotic agents as the symbolic execution tool *SEECER* and the CPL-to-IPVL compiler in C++.

As a case study, we consider two CPL plans acting on the Wumpus World (see Section II-D). Our primary verification objective is to ensure the safety of the plan execution. All experiments are performed on a Linux machine with a 3.5 GHz Intel processor using the Z3 SMT solver [20] (version 4.8.0). In the following we describe our two plans (Section IV-A), the verification annotations (Section IV-B) and the results of the experimental evaluation (Section IV-C) in more detail.

A. CPL Plans on the Wumpus World

We developed two plans with different complexity acting on the Wumpus World. While certainly not optimal in terms of finding the gold, we expect both plans to be safe, i. e. the agent will never die due to a pit or Wumpus. To investigate the bug-finding capabilities of SEECER, we also consider earlier faulty versions of both plans.

1) Slalom Plan: This plan explores the dungeon in a slalom pattern, starting by walking north. Upon perceiving a glitter, the agent will grab the gold and leave the dungeon by walking back to room $(0, 0)$ on the same path and eventually climbing out. After perceiving a stench, it will shoot its arrow. If the agent has no arrow left or perceives a breeze, it will also leave the dungeon without further exploration. In its faulty version, the plan chooses an incorrect path when leaving the dungeon, potentially sending the agent through unsafe territory.

2) Column-wise Plan: This plan explores more of the environment even after perceiving a stench or breeze. Similarly to the Slalom Plan, the agent will avoid taking risks by exploring potentially dangerous rooms. It will instead try to walk as far north as safely possible, then return to the southernmost room in its column, move one column to the right and repeat the same process there. The agent will also pick up the gold if it encounters a glitter. After all columns have been explored, the agent returns to room $(0, 0)$ and climbs out of the dungeon.

Example 8. *The function in Listing 8 is part of the Column-wise Plan. It is supposed to determine if a room's neighborhood is safe. The result of this function is used to guide the agent's exploration.*

The function first checks for a breeze in the current room, implicitly updating the belief state (Line 2). If it encounters a breeze, the current neighborhood is deemed unsafe (Line 3). Otherwise, it checks for a stench (Line 4). If a stench is perceived, it shoots an arrow in its current viewing direction (Line 7). After shooting, the neighborhood is labeled as safe iff the stench has vanished (Line 9).

The faulty version of this plan misses the negation in Line 9 of Listing 8. This will cause the agent to sometimes label an unsafe neighborhood as safe, which might lead to dangerous exploration.

```
1  (defun is-neighborhood-safe ()
2    (if (perform (an action (type perceive) (signal
       breeze)))
3        nil
4      (if (perform (an action (type perceive) (signal
       stench)))
5    (if has-arrow
6        (progn
7          (perform (an action (type shooting)))
8          (setq has-arrow NIL)
9          (not (perform (an action (type perceive) (
       signal stench))))))
10       T)))
```

Listing 8: Function is-neighborhood-safe

Next, we consider verification of the presented plans. Therefore, we first have to specify assertions.

B. Verification Annotations

We formulate three classes of assertions on the Wumpus World and our CPL plans. Each class corresponds to a different verification goal:

- *Safety assertions*: these assertions ensure that the agent never walks into a pit or Wumpus. These are the most important assertions, as any plan violating them puts the agent in danger. Consequently, they will also be the main focus in our evaluation.
- *Consistency assertions*: the agents belief state is compared to the environment model to check for any inconsistencies such as differing positions. Consistency assertions are particularly useful during development to avoid safety risks or unwanted behavior later on.
- *Livelock assertions*: a maximum number of actions is imposed on the agent to avoid livelocks, e. g. the agent walking in circles.

Besides the assertions, we also specify some general assumptions about the environment. More precisely we require a valid initial environment configuration, e. g. no two pits are in the same room.

C. Experimental Evaluation

For evaluation, we consider both plans as well as their faulty versions, each in combination with square Wumpus Worlds of edge lengths 3 to 10 rooms. Further, we fixed the number of Wumpus' and gold nuggets to one, but tried multiple numbers of pits (0, 1, and 5). The agent always starts in room $(0, 0)$, while the positions of Wumpus, gold and pits are fully symbolic. This enables a comprehensive plan verification for all possible environment configurations within these boundaries. Finally, we use the verification annotations described in Section IV-B.

We observed, that SEECER has been highly effective in finding the bugs in both faulty plan versions. For each combination of plan and environment setup (i. e. size of the Wumpus World and the number of included pits) SEECER

TABLE I: SEECER plan verification results

Slalom Plan: safe version									
pits		3×3	4×4	5×5	6×6	7×7	8×8	9×9	10×10
0	T	1s	3s	7s	14s	27s	46s	1m	2m
	#P	10	22	38	58	82	110	142	178
1	T	2s	5s	14s	32s	1m	2m	3m	6m
	#P	13	31	55	85	121	163	211	265
5	T	2s	7s	26s	1m	3m	5m	9m	16m
	#P	4	19	43	73	109	151	199	153
Column-wise Plan: safe version									
pits		3×3	4×4	5×5	6×6	7×7	8×8	9×9	10×10
0	T	1s	4s	11s	26s	54s	2m	3m	4m
	#P	6	13	22	33	46	61	78	97
1	T	5s	40s	3m	12m	34m	1h33m	3h39m	7h56m
	#P	21	102	306	722	1464	2670	4502	7146
5	T	1s	1m	21m	3h49m	TO	TO	TO	TO
	#P	2	115	1319	10357	—	—	—	—

T: execution time (s=seconds, m=minutes, h=hours)
#P: number of symbolic execution paths, **TO**: Timeout (8h)

found a counterexample demonstrating the bug on the CPL plan leading to unsafe behavior in less than a second. In the following, we focus on the more interesting results, namely proving safety of the bug-free plans.

Table I shows the results for the safe versions of the *Slalom* plan (upper half of Table I) and *Column-wise* plan (lower half of Table I). We report the execution time T and the number of paths $\#P$ for each combination of plan and environment setup. In order to prove desired behavior (i. e. none of the assertion classes specified in Section IV-B is violated), SEECER needs to explore the complete symbolic state space.

It can be observed, that the verification time correlates with the environment complexity. This is to be expected, as the environment model has a direct influence on the state space size. Furthermore, the verification time also depends on the actual plan. While SEECER is able to handle the *Slalom Plan* with increasing environment complexity, it can be observed that the verification runtimes grow exponentially for the *Column-wise Plan*. This can be explained with the significantly larger branching logic in the *Column-wise Plan*, which in turn leads to a much larger number of symbolic execution paths (#P) and SMT solver queries. Symbolic state merging should be a viable technique to increase the scalability of SEECER on such problem instances.

Nonetheless, despite currently missing state-of-the-art optimizations in the symbolic execution engine, the evaluation already demonstrates the applicability and effectiveness of our approach in verifying cognition-enabled robotic plans and indicates that the general approach can be a suitable foundation to deal with larger and more complex environments and plans.

V. Conclusion and Future Work

In this paper, we proposed the first formal verification approach for CPL plans of cognition-enabled robots. Therefore, we introduced (1) the *Intermediate Plan Verification Language* (IPVL) and (2) the *Symbolic Execution Engine for Cognition-Enabled Robotics* (SEECER), which is tailored for IPVL. Our approach compiles CPL plans in combination with environment models and verification annotations down to a combined IPVL description and then applies SEECER to check for assertion violations. Our case study demonstrated the applicability and effectiveness of our approach in finding bugs as well as validating safety, consistency, and termination (convergence) of a plan.

For future work we want to: (1) investigate modeling of larger and more complex environments such as households, (2) take the interaction between the plan and knowledge base of the robot into account, and (3) integrate state merging and other state-of-the-art symbolic execution techniques, like compiled symbolic simulation [21], into SEECER to improve scalability.

References

[1] B. Drabble, "EXCALIBUR: a program for planning and reasoning with processes," *Artif. Intell.*, vol. 62, no. 1, pp. 1–40, 1993.

[2] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proc. of the IEEE*, vol. 91, no. 1, pp. 212–237, 2003.

[3] M. Beetz, L. Mösenlechner, and M. Tenorth, "Cram—a cognitive robot abstract machine for everyday manipulation in human environments," in *IROS*. IEEE, 2010, pp. 1012–1017.

[4] D. V. McDermott, "A temporal logic for reasoning about processes and plans," *Cogn. Sci.*, vol. 6, no. 2, pp. 101–155, 1982.

[5] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia, "Automated composition of motion primitives for multi-robot systems from safe LTL specifications," in *IROS*, 2014, pp. 1525–1532.

[6] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artif. Intell.*, vol. 116, no. 1-2, pp. 123–191, 2000.

[7] P. Doherty and J. Kvarnström, "Talplanner: A temporal logic-based planner," *AI Mag.*, vol. 22, no. 3, pp. 95–102, 2001.

[8] E. D. Sacerdoti, "The nonlinear nature of plans," Stanford Research Inst Menlo Park CA, Tech. Rep., 1975.

[9] K. Erol, J. Hendler, and D. S. Nau, "Htn planning: Complexity and expressivity," in *AAAI*, vol. 94, 1994, pp. 1123–1128.

[10] R. Bellman, "On the theory of dynamic programming," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 38, no. 8, p. 716, 1952.

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

[12] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "Golog: A logic programming language for dynamic domains," *J. Logic Prog.*, vol. 31, no. 1-3, pp. 59–83, 1997.

[13] J. Claßen and G. Lakemeyer, "On the verification of very expressive temporal properties of non-terminating Golog programs," in *ECAI*, 2010, pp. 887–892.

[14] F. Baader and B. Zarrieß, "Verification of golog programs over description logic actions," in *FroCoS*. Springer, 2013, pp. 181–196.

[15] D. Araiza-Illan, D. Western, A. Pipe, and K. Eder, *Coverage-Driven Verification — An Approach to Verify Code for Robots that Directly Interact with Humans.*, 2015, pp. 69–84.

[16] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[17] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[18] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM, Comp. Surveys*, vol. 51, no. 3, p. 50, 2018.

[19] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[20] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340, available at https://github.com/Z3Prover/z3.

[21] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.