MODELLING, VERIFICATION AND TEST OF HIGH-LEVEL ROBOTIC PLANS

TIM JANIS MEYWERK

A Dissertation Submitted to
The Faculty of Mathematics and Computer Science
University of Bremen

In Partial Fulfilment of the Requirements for the Academic Degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Supervisor and primary reviewer:
Prof. Dr. Rolf Drechsler
Group of Computer Architecture
University of Bremen

Secondary reviewer:
Prof. Michael Beetz, PhD

12 April 2023

EIDESSTATTLICHE VERSICHERUNG

Hiermit versichere ich, dass ich

1. die Arbeit ohne unerlaubte fremde Hilfe angefertigt habe,
2. keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt habe und
3. die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.
4. Die zu Prüfungszwecken beigelegte elektronische Version ist mit der abgegebenen gedruckten Version identisch.

Bremen, 12 April 2023

Tim Janis Meywerk

ABSTRACT

Robots are an integral part of current industrial processes. Typical industrial robots are used in different factory settings to handle repetitive tasks and thus ease the workload for human workers. Recent advances in technology and artificial intelligence opened the door for a generation of more mobile, flexible and autonomous robots for a wider variety of applications.

The combination of dynamic environments, complex tasks and a need for explainability call for a structured, high-level approach to autonomous robotics. One such approach is plan-based robotics, where a high-level plan is responsible for the orchestration and supervision of lower-level modules such as a motion planner, knowledge base or computer vision module. The plan itself is written in a high-level plan language.

With the increasing complexity of robotic plans, programming errors and oversights only become more likely. There is an undeniable need for a high level of safety, robustness and correctness in autonomous robots. This requires not only a thorough engineering of the robotic software, but also techniques to assess the correctness, and to uncover hidden bugs.

In this thesis, we aim to extend the state-of-the-art in verification techniques for high-level robotic plans. We present both formal and test-based methods. In particular, we make contributions to three areas of robotic plan verification. First, we present several techniques for the symbolic verification of high-level robotic plans. Our second contribution is the development of two approaches that aid in the modelling of robotic environments and thus facilitate both the planning and verification process. Finally, we present coverage-guided fuzzing as an automatic, test-based method for bug-finding in robotic plans. All of our contributions are applied to the CRAM Plan Language (CPL). They are described in detail and experimentally evaluated to demonstrate their effectiveness.

KURZFASSUNG

Roboter sind ein integraler Bestandteil aktueller industrieller Prozesse. Typische industrielle Roboter werden an verschiedenen Stellen in Fabriken eingesetzt, um repetitive Aufgaben zu bearbeiten und so das Arbeitspensum menschlicher Arbeiter zu verringern. Durch technologische Fortschritte, vor allem im Bereich künstlicher Intelligenz, wird der Weg für eine neue Generation mobilerer und flexiblerer autonomer Roboter geebnet, die für eine größere Menge von Anwendungen eingesetzt werden können.

Die Kombination aus dynamischen Umgebungen, komplexen Aufgaben und dem Bedürfnis, das Verhalten der Roboter erklären zu können, verlangen nach einem strukturierten, abstrakten Ansatz zur Steuerung autonomer Roboter. Ein solcher Ansatz ist die planbasierte Robotik, in der ein abstrakter Plan für die Koordination und Überwachung kleinerer Module wie etwa eines Bewegungsplaners, einer Wissensdatenbank oder eines Bilderkennungsmoduls verantwortlich ist. Der Plan selbst ist in einer abstrakten Plansprache geschrieben.

Mit der steigenden Komplexität der robotischen Pläne werden Programmier- und Flüchtigkeitsfehler immer wahrscheinlicher. Es gibt einen unbestreitbaren Bedarf an einem hohen Grad von Sicherheit, Robustheit und Korrektheit der autonomen Roboter. Hierzu ist nicht nur eine sorgfältige Entwicklung der Software des Roboters nötig, sondern auch Techniken, um die Korrektheit zu beurteilen und versteckte Fehler aufzudecken.

Diese Dissertation zielt darauf ab, den Stand der Forschung im Bereich der Verifikationstechniken für abstrakte Roboterpläne wesentlich zu erweitern. Hierzu werden sowohl formale als auch test-basierte Methoden vorgestellt. Insbesondere werden Beiträge zu den folgenden drei Themengebieten der robotischen Planverifikation geleistet. Der erste Beitrag umfasst mehrere Techniken zur symbolischen Verifikation von abstrakten Roboterplänen, der zweite ist die Entwicklung von zwei Ansätzen, die bei der Modellierung von robotischen Umgebungen unterstützen und somit sowohl den Planungs- als auch den Verifikationsprozess erleichtern und zuletzt wird abdeckungsgetriebenes Fuzzing als eine automatische, testbasierte Methode zur Fehlerfindung in robotischen Plänen vorgestellt. Alle vorgestellten Beiträge werden auf die CRAM Plan Language (CPL) angewendet. Sie werden in dieser Dissertation im Detail erläutert und experimentell evaluiert, um ihre Wirksamkeit zu demonstrieren.

ACKNOWLEDGEMENTS

This thesis is the result of not only my own work, but also the support of great people. I would like to use this opportunity to thank them.

I would like to express my deepest gratitude towards my advisor, Prof. Dr. Rolf Drechsler, for providing me with this opportunity and for his valuable advise and support.

I am extremely grateful to Prof. Dr. Michael Beetz for his advise and for initiating the EASE project that this thesis builds upon. I am delighted that he agreed to take the time to review this thesis.

Many thanks go to Prof. Dr. Daniel Große and Dr. Vladimir Herdt for their advise and their extensive expertise on formal verification.

I had the great pleasure to work with wonderful colleagues at AGRA and DFKI. I wish to especially thank Dr. Marcel Walter, my partner during the first years of EASE, for his decisive role in the development of SEECER and hundreds of interesting discussions on robot verification, SMT solving and FCN.

I would also like to thank Gayane Kazhoyan for her introduction to CRAM and for answering all of my CRAM-related questions. Thanks also go to Arthur Niedzwiecki for his help.

Teaching has been a passion during my time at AGRA. I am therefore very happy that I was able to share my interest in plan verification with some of my students. I am very grateful to Jan Kleinekathöfer, who implemented substantial parts of SEECER and for his valuable suggestions. I would also like to acknowledge Daniel Staack, Fenja Kollasch, Arbnor Miftari and Till Schlechtweg for their code contributions and Jonas Dech for his CRAM support.

Finally, I would like to thank my wife Jil, not only for her great suggestions, but most importantly for her continued love, encouragement and support.

The research reported in this thesis has been partially supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 Project-ID 329551904 “EASE - Everyday Activity Science and Engineering”, University of Bremen (http:/www.ease-crc.org/). The research was conducted in subproject P04.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Robots are an integral part of current industrial processes. Typical industrial robots are used in different factory settings to handle repetitive tasks and thus ease the workload for human workers. Their main purpose is to make high-volume production cheaper. These robotic applications require very limited autonomy and adaptability. However, advances in technology and artificial intelligence opened the door for a generation of more mobile, flexible and autonomous robots for a wider variety of applications.

The higher degree of autonomy enables robotic agents to work in environments that are inaccessible to humans, such as underwater environments [16, 85], urban search and rescue scenarios [27, 61] or space exploration [34, 112]. In these kinds of environments, only a limited amount of human intervention is possible, so the correctness and robustness of the robots' control programs is vital for their success.

Additionally, robots are driven to act in closer interaction and collaboration with humans in our everyday lives. These applications range from entertainment [1] to household assistance [32, 47, 106, 119], elderly care [111] and even medical procedures [40, 41, 99]. Advances in the robots' autonomy have also had influence on the manufacturing domain. The next generation of factory robots is expected to perform their tasks more autonomously and flexibly [33, 82]. In fact, autonomous robots have been identified as one of the major drivers for the Industry 4.0 [4, 45, 108].

The close contact between humans and robots that is inevitable in these applications raises concerns about how an effective and safe interaction between humans and robots can be achieved. Solutions are investigated in the fields of *Human Robot Interaction* (HRI) and *Human Robot Collaboration* (HRC) [20, 113]. The tighter interaction between humans and robots also raises ethical [3, 110, 118] and legal concerns [13, 24]. These issues generally discourage the exclusive use of black-box approaches such as machine learning, since the robots' actions should ideally be explainable and reproducible.

When autonomous robots and humans act in close proximity, accidents are bound to happen. Fortunately, publicly known accidents involving autonomous robots are rare. One incident that attracted major attention in 2018 was a collision between a self-driving car and a pedestrian who later died from her injuries [58]. In another incident, an autonomous security robot ran into and lightly injured a toddler at a shopping mall [115]. However, not all robot accidents are reported in the press. A study [5] on robot malfunctions in the medical field found 144 deaths linked to malfunctioning robot systems. A recent case study [116] on robot accidents also concludes that "the likelihood and scope of robot accidents are much greater" when robots act in human environments and calls for strict regulations regarding accident investigation. While fatal accidents should of course be avoided by any means, there are

also less dramatic malfunctions with severe consequences. For instance, a stuck autonomous robot in an inaccessible environment may cause a failed mission and potentially a loss of the robot hardware.

The combination of dynamic environments, complex tasks and a need for explainability call for a structured, high-level approach to autonomous robotics. One such approach is *plan-based robotics*, where a high-level *plan* is responsible for the orchestration and supervision of lower-level modules such as a motion planner, knowledge base or computer vision module. The plan itself is written in a high-level plan language. There are two main types of languages used for this purpose. The first are logic-based languages such as *GOLOG* [57] and its derivatives [29, 38]. The second type are extensions of general programming languages, such as TDL [97], which is an extension of C++, and RPL [69] or CPL [14], which are based on Lisp. Some languages are also not directly based on a common programming language, but are still designed in the style of a programming language. Examples include STRIPS [30], ADL [83] and PDDL [36].

The majority of high-level plans is written by hand or generated from informal descriptions [101, 102]. This obviously leaves room for programming errors and oversights. With the increasing complexity of robotic plans, these only become more likely. There is an undeniable need for a high level of safety, robustness and correctness in autonomous robots. This requires not only a thorough engineering of the robotic software, but also techniques to assess the correctness, and to uncover hidden bugs. The current state-of-the-art in autonomous robot testing is predominantly based on simulation and some isolated test runs on the real system. Cases to be simulated are usually picked by hand. This often results in an insufficient coverage of the robot software due to the high workload involved in a complete test set. Furthermore, important edge cases are often missed by engineers, especially when the software becomes more complex. In this thesis, we therefore investigate systematic methods to test and verify high-level plans for autonomous robots.

We aim to extend the state-of-the-art in verification techniques for high-level robotic plans. We present both formal and test-based methods. Most current work on robot plan verification focusses on internal properties of the plan. In this thesis, we want to additionally consider a model of the robot's environment. This enables us to also verify the plan with respect to properties regarding the environment and thus acquire more meaningful verification results. Also, unlike many other approaches, we do not restrict the plan language to simplify the verification problem. Instead, we choose the Turing-complete *CRAM Plan Language* (CPL) from the *Cognitive Robot Abstract Machine* (CRAM) framework as our plan language under verification.

Since modelling of the robotic system and its environment plays a crucial part in many of our presented approaches, we also work on techniques to assist in these modelling tasks. The modelling formalisms used here are well-known, but their application to verification tasks and their automatic generation and debugging is a novel research direction.

We also present coverage-guided fuzzing as a test-based method for robotic plan verification. Although some work has already been done on fuzzing in robotics, existing approaches are usually not based on coverage and do not incorporate the robots' environment.

1.1 THESIS OUTLINE

This thesis considers several approaches to enable the verification of high-level plans for autonomous robots. These include

- symbolic verification of robotic plans,
- formal methods for modelling assistance, and
- coverage-guided fuzzing of robotic plans.

We outline each of these topics in more detail in the remainder of this section.

SYMBOLIC VERIFICATION OF ROBOTIC PLANS (CHAPTER 4)
Symbolic verification replaces concrete inputs to a computer program with symbolic variables. Executing a program in this way yields symbolic constraints over the program inputs, variables and outputs, which can then be used by a constraint solver to find execution traces that e. g. violate safety properties. As a first step, we adapt symbolic execution to the domain of plan-based robotics and present a first approach to symbolic execution of CPL plans under a set of assumptions and assertions. To this end, we present a symbolic execution engine for CPL and a methodology to integrate environment models written in Common Lisp into the verification process. Secondly, we expand our approach to environments modelled in a logical formalism, namely the *Discrete Event Calculus* (DEC). We present a pure DEC reasoning approach to verify simple action sequences and then integrate this with our symbolic execution engine for the verification of complex CPL plans. Thirdly, we take a closer look at CPLs failure handling capabilities and present a technique, based on our symbolic execution engine, to find low-level failures that are not properly handled by the high-level plan. This approach is agnostic of the concrete environment and instead assumes that any action may fail at any time, yielding a complete list of all unhandled low-level failures.

FORMAL METHODS FOR MODELLING ASSISTANCE (CHAPTER 5) Formal models of a robotic environment are the backbone of decades of work in the planning domain. In Chapter 4 we also use formal models for the verification of plans. Models are supposed to be abstract, simple and usually discrete, while still accurately modelling the complexity and continuity of the real world. This is a major challenge for model designers. We therefore propose tools to support model designers during this task. First, we propose a novel methodology for SMT($\mathcal{LRA}$) learning, which may be used to automatically build SMT($\mathcal{LRA}$) formulae from a given set of examples. These formulae

could then be used to divide the continuous space of the robotic environment into meaningful discrete regions. Secondly, we present a technique for the debugging process of formal models. Here, we use our symbolic execution engine to automatically find executions in which the behaviour of the formal model differs from that of a simulation engine.

COVERAGE-GUIDED FUZZING OF ROBOTIC PLANS (CHAPTER 6) Formal verification methods are a great way to ensure completeness of the verification process. This does however come with the drawback of high runtimes and poor scalability. Practically speaking, this results in a failure of formal verification for complex plans and environments. Therefore, we also consider test-based methods for the verification of robotic plans, in particular coverage-guided fuzzing. We introduce coverage-guided fuzzing to the domain of plan-based robotics and present a prototypical implementation for CPL. In addition, we introduce a novel coverage metric tailored for the domain of plan-based robotics.

1.2 PUBLICATIONS

This thesis is based on several peer-reviewed conference publications. They have been incorporated in this thesis as follows:

- Section 4.1: Symbolic Execution of Robotic Plans – [M1]
- Section 4.2: Verification via Logic-based Environment Modelling – [M2]
- Section 4.3: Symbolic Fault Injection for Robotic Plans – [M3]
- Section 5.1: Clustering-guided SMT($\mathcal{LRA}$) Learning – [M4]
- Section 5.2: Simulation-based Debugging of Formal Environment Models – [M5]
- Chapter 6: COVERAGE-GUIDED FUZZING OF ROBOTIC PLANS – [M6]

A list of all of the author's publications related to the topic of this thesis is given below:

[M1] Tim Meywerk, Marcel Walter, Vladimir Herdt, Daniel Große, and Rolf Drechsler. "Towards Formal Verification of Plans for Cognition-Enabled Autonomous Robotic Agents". In: *Conference on Digital System Design (DSD)*. 2019, pp. 129–136.

[M2] Tim Meywerk, Marcel Walter, Vladimir Herdt, Jan Kleinekathöfer, Daniel Große, and Rolf Drechsler. "Verifying Safety Properties of Robotic Plans Operating in Real-World Environments via Logic-Based Environment Modeling". In: *International Symposium on Leveraging Applications of Formal Methods (ISoLA)*. 2020, pp. 326–347.

[M3] Tim Meywerk, Vladimir Herdt, and Rolf Drechsler. “Symbolic Fault Injection for Plan-based Robotics”. In: *International Conference on Control Automation and Systems (ICCAS)*. 2022, pp. 1710–1715.

[M4] Tim Meywerk, Marcel Walter, Daniel Große, and Rolf Drechsler. “Clustering-Guided SMT($\mathcal{LRA}$) Learning”. In: *International Conference on Integrated Formal Methods (IfM)*. 2020, pp. 41–59.

[M5] Tim Meywerk, Arthur Niedzwiecki, Vladimir Herdt, and Rolf Drechsler. “Simulation-Based Debugging of Formal Environment Models”. In: *Mediterranean Conference on Control and Automation (MED)*. 2022, pp. 890–895.

[M6] Tim Meywerk, Vladimir Herdt, and Rolf Drechsler. “Coverage-guided Fuzzing for Plan-based Robotics”. In: *International Conference on Agents and Artificial Intelligence (ICAART)*. accepted for publication. 2023.

2

PRELIMINARIES

To keep this thesis self-contained, this chapter reviews important foundations and concepts necessary for the understanding of the remainder of the thesis. Section 2.1 introduces the Cognitive Robot Abstract Machine, which is the main objective of the verification approaches presented in this thesis. Section 2.2 reviews several verification techniques that this work extends and adapts to the domain of high-level robotic plans.

2.1 COGNITIVE ROBOT ABSTRACT MACHINE

The *Cognitive Robot Abstract Machine* (CRAM) [14, 74] is a framework for the implementation of *cognition-enabled* robotic plans. The term cognition-enabled here describes a system that is able to reason about its actions and provide explanations of that reasoning. CRAM features the *CRAM Plan Language* (CPL) for the description of the robot's behaviour.

Fig. 1 shows a CRAM-centric view of the architecture of the CRAM ecosystem and its interaction with the environment. The centrepiece of the architecture is the CRAM executive, which executes a plan written in CPL. CPL is the high-level plan language used throughout this thesis and will be explained in more detail in Section 2.1.1.

To reason about the world, the CRAM executive is able to query the robot's belief state and an ontology-based knowledge base. Information about the world acquired through the robot's perception module is also fed back into the belief state.

A main concept of CPL is the use of *designators*. A designator is an abstract description of an action, object or location. For instance, an action designator could describe the action of grasping a green object. The corresponding designator would however leave other properties of the object and action such as the concrete trajectory unspecified. The *designator dereferencing* module is responsible for filling in the remaining parameters and thus producing an executable action description. The dereferencing is achieved through a combination of hand-written heuristics, machine learning and simulations. For the latter, a fast projection simulator is used to try different parametrizations in a simulated environment. We also use this simulator in some parts of this thesis and will therefore introduce it in more detail in Section 2.1.3.

The dereferenced designators are then used to interact with the environment. Action designators in particular are used to perform perception, navigation and manipulation actions to perceive objects, move within the environment and manipulate objects, respectively. Each of the three tasks has its own module activated by the CRAM executive.

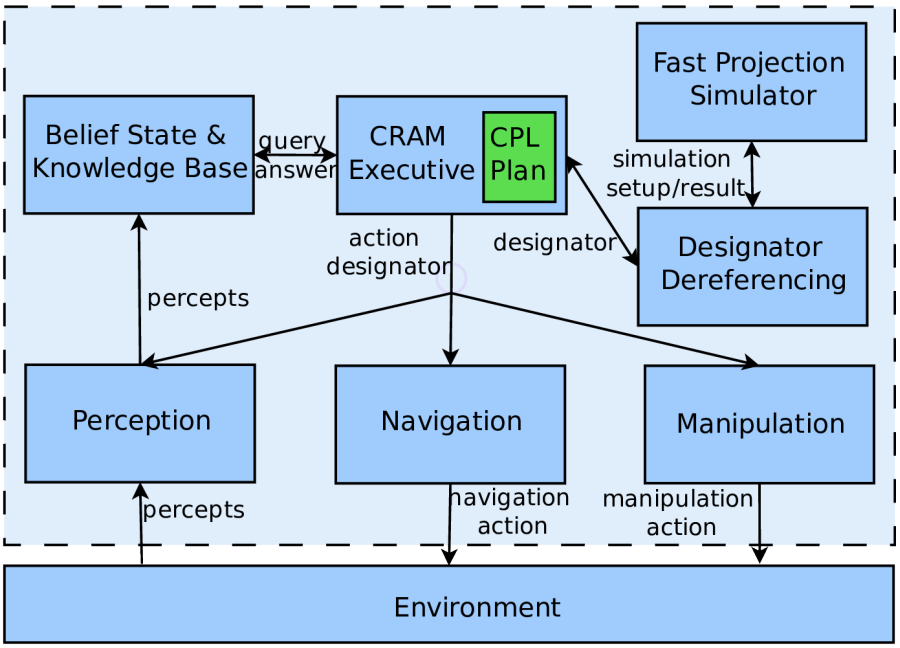


Figure 1: Overview of the CRAM framework

```
1 (let* ((a 10) (b 12))
2   (if (> a 0)
3     (* 3 (+ a b))
4     (* 3 b)))
```

Figure 2: Common Lisp example

2.1.1 *CRAM Plan Language*

Writing plans for autonomous robots in multiple changing environments for various tasks requires a general and flexible plan language. In this section, we introduce the plan language of CRAM, CPL.

CPL is an extension of the Common Lisp programming language. Therefore, we will shortly review Common Lisp's structure before proceeding with CPL.

Definition 1. *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 Fig. 2. It consists of 9 non-atomic s-expressions, which are nested and perform simple arithmetic*

```
1 (let*
2         ((object-desig (an object (type cup)))
3         (location-desig (a location (on table)))
4         (action-desig (an action (type placing) (
              object object-desig) (target
              location-desig))))
5         ...)
```

Figure 3: Designators in CPL

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.*

Common Lisp is generally not an object-oriented language. However, classes and objects can be used through the *Common Lisp Object System* (CLOS). CPL also makes extensive use of this functionality, e. g. for designators.

As CPL is an extension of common Lisp, all of Common Lisps built-in functions can be used in CPL as well. The most important functions and macros in the context of robotic planning are however only available in CPL.

Designators play an essential role in the interface between CPL and the robot's environment. They allow to specify actions, objects and locations in an abstract, incomplete way. In CPL, designators are constructed using the `a` and `an` macros. They take the type of designator (`action`, `object` or `location`) as a first parameter, followed by a number of key-value pairs to describe the underlying action, object or location.

Example 2. *Consider the code snippet in Fig. 3. The CPL code creates three designators and assigns them to three local variables. The object designator* `object-desig` *describes a cup, without specifying the concrete cup or other properties such as the cup's dimensions, colour or position. Similarly, the location designator* `location-desig` *describes a location on top of a table, but leaves the exact position open. The action designator* `action-desig` *now describes the action of placing the object of type cup on some location on the table.*

To transform a designator into a concrete action, object or location, it has to be dereferenced using the `reference` function. This function can be called explicitly on each designator, but it is also called whenever an action designator is executed or an object or location designator is used as a parameter to an executed action. Actions are executed using the `perform` function.

Example 3. *Consider again the designators from the previous example. The placing action can be executed through* `(perform action-desig)`*. Only then the abstract descriptions of the cup, table-top position and placing actions are dereferenced and replaced with a concrete object, position and arm trajectory.*

In Common Lisp, every function and macro must have a return value. This is also true for CPL's `perform` function. For most action types, this return

```
1 (with-failure-handling
2         ((failure-type (e)
3             handler))
4       body)
```

Figure 4: CPL failure handling

value is `nil` and is usually ignored by the plan. Some other actions will however have meaningful return values. For instance, the `perceiving` action returns a designator for the object it has perceived.

Another important feature of CPL is its extensive failure handling capability to deal with any unforeseen events caused by the highly dynamic environments of autonomous robots. Failure handling is initiated through the `with-failure-handling` macro. Fig. 4 shows the general structure of the macro. The *body* of the macro is executed first. Whenever a failure of type *failure-type* occurs during execution of the body, the *handler* is called. This handler will then try to remove the cause of the failure. Multiple failure handling macros can be used inside each other, creating a hierarchy of failure handlers. If a handler is not able to remove the failures root cause, it may choose to ignore it, rethrow it to a higher level or throw a new failure of a different type instead.

2.1.2 *CLisp Bytecode*

There are several implementations for the Common Lisp programming language. Most of these are full compilers, i. e. they compile the Common Lisp source directly into machine language. The CLisp compiler [39] follows a different approach by compiling Common Lisp into its own bytecode representation, which is then interpreted. Since the bytecode produced by CLisp is simpler and easier to analyse than both the original code and machine language, we build most of our techniques presented in this thesis on the bytecode representation. In this section, we introduce the syntax and semantic of a human-readable version of the CLisp bytecode.

The bytecode is organized in a set of functions. Each function is composed of a data section and a code section. The data section is a list of constants to be used inside the function. These can be constant values of any of Common Lisps built-in data types, lists or names of other functions or variables. The code section is a sequence of instructions with parameters. Functions are either top-level functions or callable functions. When the bytecode is interpreted, all top-level functions are executed in the order in which they appear in the code. Callable functions are not executed directly, but may instead be called inside the code section of another function.

The CLisp bytecode is executed on a virtual stack machine. It consists of a value stack, which stores most of the values computed during execution, a function stack to trace the currently executed functions and return addresses, a map of global variables and their values, and the single register `last_value`

storing the result of the last executed instruction. The following paragraphs describe the instruction categories that are of particular importance to this thesis.

Constant instructions load constant values into the `last_value` register. These can be either the truth values `T` and `NIL` or some constant from the data section of the function.

Variable instructions load a value from a global variable into `last_value` or store `last_value` in a new or existing global variable.

Stack instructions push the value stored in `last_value` onto the value stack or remove the top element from the stack. Most instructions that compute new values are immediately followed by a `PUSH` instruction to push the result onto the stack.

Control flow instructions are used to implement conditions, loops and returns. The most important control flow instructions are jump instructions, which set the next instruction to be executed. Jumps can either be unconditional such as the `JMP` instruction, or conditional such as the `JMPIFNOT` instruction. The latter will only jump to its target address if `last_value` contains the value `NIL`.

The actual computations are performed using instructions from the *call* category. These are function calls to either user-defined or built-in Common Lisp functions. There are different instructions such as `CALL1` or `CALL2` depending on the number of arguments. For a function with n arguments, the top n elements from the value stack are consumed, removing them from the stack.

Other categories of bytecode instructions are omitted here, since they occur only rarely and have no special importance in the context of this thesis.

Example 4. *Consider the bytecode function in Fig. 5. The first four unnumbered lines represent the data section. Here the numeral* 2*, the strings* `"ODD"` *and* `"EVEN"` *as well as the variable name* `IS-EVEN` *are provided for the function to use.*

The function expects a single argument to be present on the stack already. The code section starts at Line 1 by loading the constant at index 0*, namely the numeral* 2 *and then pushing it onto the stack. The next instruction on Line 3 calls a function with two arguments. The index of the function is* 210*, which corresponds to the built-in modulo function. Consequently, Line 3 will compute the modulo operation on the top two elements on the stack, which are the functions argument and* 2*. Here, the two elements are removed from the stack. In the next line, the result is pushed onto the stack. Now the stack contains exactly one element, which is either* 0 *or* 1*, depending on whether the original argument was even or odd. The following* `ZEROP` *function in Line 5 compares that value to zero. If the value is equal to zero (i. e. the original argument was even),* `ZEROP` *writes* `T` *into* `last_value`*, otherwise* `NIL`*. The result is then assigned to the global variable* `IS-EVEN` *in Line 6. The following* `JMPIF` *instruction also takes the result and uses it as a condition for a conditional branch. If* `ZEROP` *returns* `T`*, the execution jumps to Line 11, where the constant* `"EVEN"` *is pushed onto the stack. Otherwise the function*

```
(CONST 0) = 2
(CONST 1) = "ODD"
(CONST 2) = "EVEN"
(CONST 3) = IS-EVEN

1  (CONST 0)          ; 2
2  (PUSH)
3  (CALLS2 210)       ; MOD
4  (PUSH)
5  (CALLS1 172)       ; ZEROP
6  (SETVALUE 3)       ; IS-EVEN
7  (JMPIF L11)
8  (CONST 1)          ; "ODD"
9  (PUSH)
10 (JMP L14)
11 L11
12 (CONST 2)          ; "EVEN"
13 (PUSH)
14 L14
15 (SKIP&RET 1)
```

Figure 5: CLisp bytecode

continues at Line 8 and pushes "ODD" onto the stack. The final Line 15 terminates the function, leaving its result ("ODD" or "EVEN") as the top element of the value stack.

2.1.3 *Fast Projection Simulator*

CRAM's fast projection simulator [75] is based on the Bullet Physics engine [23]. It can simulate robots, static environment objects such as walls or furniture and manipulatable objects such as cups or bottles based on data provided in the *Unified Robot Description Format* (URDF). The goal of the simulator is a very fast, yet realistic simulation of the effect of action parametrizations. Due to a faster-than-real-time simulation, multiple different parametrizations can be simulated before deciding which one to execute on the real robot. This high execution speed is also a prerequisite for some methods presented in this thesis.

To achieve a high simulation speed, not all physical properties are simulated with perfect accuracy. For instance, movements of the robot are executed in zero time, which results in the robot teleporting to its target pose. Therefore, obstacles in the way do not necessarily stop the robot's action. This requires some additional checks, which we implemented for the scenarios described in this thesis.

The simulation can be executed in GUI mode or direct mode. In direct mode, no graphical representation of the simulated robot and environment is shown to achieve a higher simulation speed. GUI mode, on the other hand,

can be used to visualize critical scenarios for the user or for debugging purposes.

2.2 VERIFICATION TECHNIQUES

Verification is the task of determining whether a system adheres to a given specification. The specification is usually given as a set of properties that have to be fulfilled by the system. When a system violates some property, we call this an *error* or *bug*.

We distinguish between two types of verification. *Formal verification* uses mathematical reasoning to identify errors. Formal methods often enable complete reasoning, i. e. they will either find an error or prove that no errors exist with respect to the given specification. This completeness often comes with the price of a high runtime of the verification process and depending on the type of the system or properties, the underlying problems may be undecidable altogether. *Test-based verification*, on the other hand, tries to find errors by feeding the system with different inputs and comparing the results to expected values. Unlike formal verification, test-based methods are unable to prove the absence of errors, but usually offer much better scalability.

This section reviews some verification techniques that are used and extended within this thesis. These are satisfiability modulo theories in Section 2.2.1, symbolic execution in Section 2.2.2, fault injection in Section 2.2.3 and coverage-guided fuzzing in Section 2.2.4.

2.2.1 *Satisfiability Modulo Theories*

The *Boolean satisfiability problem* (SAT) is a major building block for many verification systems. The wide usage of SAT has been facilitated by major advances in SAT solving techniques [67, 86]. These allow SAT solvers to formally verify industrial-scale systems.

Satisfiability modulo theories (SMT) is an extension of SAT that allows to add formulae from other theories such as linear arithmetic or bitvector arithmetic to the SAT formula. This allows for a wider range of applications, e. g. software verification, but also increases the difficulty of the solving process.

In this section, we will review the formulation of SAT and SMT instances and present some theories used in this thesis.

Definition 2. *A SAT instance is a Boolean formula in* conjunctive normal form *(CNF). A CNF is a conjunction of* clauses *and each clause is a disjunction of* literals*. A literal is either a Boolean variable or its negation. The problem of SAT is to determine if a given SAT instance is* satisfiable*. A Boolean formula is satisfiable iff there is at least one assignment from Boolean variables to truth values* true *(*$\top$*) and* false *(*$\bot$*), such that the whole formula evaluates to* $\top$*.*

SMT extends upon the SAT problem in two ways. Firstly, SMT instances do not need to be in CNF, but can instead use Boolean operators in an arbi-

trary way. This does not yet increase the difficulty of the problem, since any Boolean formula can be transformed into an equisatisfiable CNF in polynomial time. Secondly, SMT allows to use constraints from theories other than Boolean logic in place of Boolean literals. The most important theories in this thesis are the arithmetic theories of real numbers and integers.

Definition 3. *A* term *in real (integer) arithmetic is defined as follows. Any real (integer) variable is a term. Any real (integer) constant is a term. If t is a term, then $-t$ is also a term. If t_1 and t_2 are terms, then $t_1\ t_2$, $t_1 - t_2$, $t_1 \cdot t_2$ and t_1/t_2 are also terms. If t_1 and t_2 are terms, then $t_1 < t_2$, $t_1 \leq t_2$, $t_1 > t_2$, $t_1 \geq t_2$ and $t_1\ \ t_2$ are* constraints*, which can be used in place of Boolean literals in the SMT instance. The semantics of terms and constraints follow the usual meaning of the arithmetic symbols.*

Unrestricted arithmetic constraints generally make the resulting SMT problem undecidable. Still, in many practical cases a satisfying assignment or unsatisfiability proof can be found. If one wants to guarantee a conclusive result however, the arithmetic constraints need to be restricted. A common way to achieve this is the restriction to linear constraints.

Definition 4. *A linear real (integer) constraint is of the form $a_1 \cdot r_1\ \cdots\ a_n \cdot r_n \leq d$, where r_i are real (integer) variables and a_i and d are real (integer) constants.*

The satisfiability problem for SMT formulae with only linear constraints is NP-complete, the same as SAT.

Example 5. *Consider the Boolean variable b and the real variables r_1 and r_2. The formula $-0.5 \cdot r_1\ \ 2 \cdot r_2 \leq 1 \wedge r_1 \cdot r_2\ \ 4 \vee \neg b$ is an SMT instance. Here, $-0.5 \cdot r_1\ \ 2 \cdot r_2 \leq 1$ is a linear real constraint and $r_1 \cdot r_2\ \ 4$ is a non-linear constraint.*

To indicate the type of constraints that are used in a problem, different abbreviations are added to the SMT acronym. For instance, the subset of SMT with only the theory of linear real arithmetic is known as SMT($\mathcal{LRA}$).

The satisfiability of SMT formulae can be determined by specialized SMT solvers such as the Z3 solver [25].

2.2.2 *Symbolic Execution*

Symbolic Execution [49] is a technique for formal software verification and bug finding. It analyses the behaviour of a program pathwise by treating inputs as symbolic values.

Symbolic execution manages a set of *execution states*. These contain a mapping of program variables to symbolic expressions constraining the value of the variable. Additionally, each execution state has a Boolean path condition *pc* describing the constraints that need to be satisfied to reach the current path of the program. Initially, the path condition is set to $\top$, i. e. there are no constraints on the execution state.

```
1  int abs(int num){
2          int result;
3          if (num < 0){
4                  result = -num;
5          }else{
6                  result = num;
7          }
8          assert(result >= 0);
9          return result;
10 }
```

Figure 6: A simple C method

Along an execution path s, the program state is updated according to the execution semantics of each instruction.

An assignment instruction overwrites the value of a variable with the right-hand-side expression. Whenever variables are used on the right-hand side of an assignment, their values from the mapping are substituted.

At each branch instruction, the execution path s is *split* into two independent paths $s_\top$ and $s_\bot$ due to two possible evaluations of the branch condition c. The pc for each path is updated accordingly as $pcs_\top \quad pcs \wedge c$ and $pcs_\bot \quad pcs \wedge \neg c$, respectively. Only feasible paths will be explored further. A path is feasible iff its pc is satisfiable.

For verification purposes, two additional instructions can be used. To add assumptions about inputs and variables, *assume(c)* adds c to the current pc. This way, irrelevant paths can be pruned, speeding up the verification process and avoiding false positives. The verification properties are expressed through *assert(c)* statements. Whenever an assert statement is encountered, it is checked whether c can be violated under the current path condition, i. e. $pc \wedge \neg c$ is satisfiable.

All constraints can be expressed as SMT instances, so that an SMT solver can be used for the satisfiability checks.

Example 6. *Consider the simple C method in Fig. 6, which computes the absolute value of its input. When this method is symbolically executed while treating* `num` *as a symbolic value, the execution state is split at the* `if` *statement in Line 3. The two resulting states have a path condition of* $\texttt{num} < 0$ *and* $\texttt{num} \geq 0$*. In the first case, the value of* `result` *is mapped to* $-\texttt{num}$ *and in the second case to* `num`*. At the assert statement in Line 8 the assertion is checked for possible violations. For the first execution state this corresponds to an SMT instance of* $\texttt{num} < 0 \wedge \neg -\texttt{num} \geq 0$ *and in the second state to* $\texttt{num} \geq 0 \wedge \neg \texttt{num} \geq 0$*. Both instances trivially evaluate to* $\bot$ *for all values of* `num`*, therefore proving the correctness of the* `abs` *method with respect to the assertion.*

A more comprehensive overview on symbolic execution and possible variations and optimizations can be found in [11, 19].

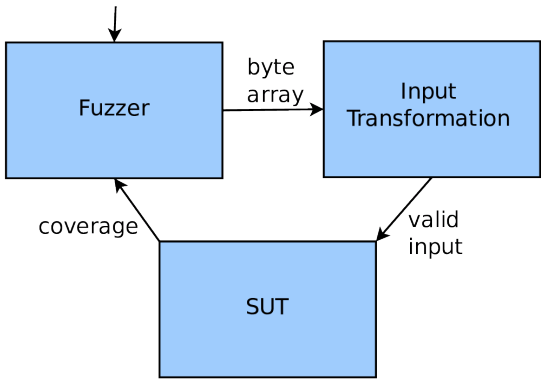


Figure 7: General coverage-guided fuzzing flow

2.2.3 *Fault Injection*

In many cases, the correct behaviour of a program under normal circumstances can be verified through test cases or formal methods. However, a challenging problem is determining how a system reacts to faults that are outside of its control, e. g. ones that stem from an external library. One way to analyse a systems robustness to faults is *fault injection*.

Fault injection has a long tradition in several different application areas to perform robustness evaluations. As such it has been leveraged at the hardware level to induce faults into netlists [2], RTL descriptions [103] or even system-level models [100] as well as at the software level [52].

Symbolic fault injection [56] extends upon the idea of traditional fault injection by using non-deterministic locations for the injection and hence enables to produce complete coverage of the system under verification with guarantees regarding its robustness. As such symbolic fault injection is a very powerful technique for finding gaps in the failure handling of complex systems. Symbolic fault injection has been mostly applied to embedded systems [56, 81]. To the best of our knowledge, this thesis is the first to use symbolic fault injection for plan-based robotics.

2.2.4 *Coverage-Guided Fuzzing*

Fuzzing [71] is a technique for software testing, which originated in the security domain and has since been applied to several different applications such as memory safety [31], network protocols [37] or hardware/software co-verification [18].

Fuzzing can be described as an interplay between the *system under test* (SUT), which is usually a program or function with an input, and a *fuzzer*. The fuzzer generates (semi-)random inputs to the SUT. The generation may be either fully random or guided by some policy or metric. When the code coverage is used to guide the fuzzing process, it is referred to as *coverage-guided fuzzing*.

The usual flow is shown in Fig. 7. The fuzzer starts by generating a random byte array. This byte array is then transformed into valid inputs to the SUT. Depending on the complexity of the input, this transformation can range from a straightforward reinterpretation to an elaborate construction of nested objects or files.

Once a valid input to the SUT has been formed, the SUT is executed. During execution, the code coverage is measured and fed back to the fuzzer. In subsequent iterations, the fuzzer will modify its input byte array either by adding or removing bytes or by mutating existing ones. The coverage can be used to decide which modifications of the byte array have been particularly successful and thus use those more often. Usually, the byte array produced by the fuzzer will start small and grow over time, producing more complex inputs the longer the fuzzing process runs.

In many implementations, the coverage will be managed using a finite amount of coverage points. Each coverage point is a point in the SUT which is of particular importance to the coverage metric. The fuzzer will then store a counter for each coverage point, indicating how often that point has been reached.

There are a large number of coverage metrics, each with its own advantages and disadvantages. They can be roughly divided into two categories. *Structural* coverage metrics depend purely on the structure of the SUT. They will analyse which parts of the source code have been executed, but will ignore the underlying semantics of the program. *Functional* coverage metrics, on the other hand, do not necessarily analyse the executed source code, but rather which of the underlying features and objectives of the SUT have been executed. They are therefore highly domain-specific.

Two examples for structural coverage metrics used in this work are the *instruction coverage* and the *branch coverage*. Instruction coverage measures what percentage of singular instructions have been executed. Therefore each instruction corresponds to a coverage point. Branch coverage on the other hand looks at the conditional branching instructions and their outcome. To reach 100% branch coverage, each branching condition must have been evaluated to both true and false at least once. In general, this makes branch coverage a stricter metric than instruction coverage. 100% branch coverage implies that 100% instruction coverage has also been reached, while the reverse is not necessarily true.

For a comprehensive overview of fuzzing, refer to [59].

3 RELATED WORK

In this chapter, we review publications related to the topics of this thesis. These include publications concerning formal verification in Section 3.1, environment modelling in Section 3.2 and fuzzing in Section 3.3.

3.1 FORMAL VERIFICATION OF ROBOTIC PLANS

The need for provable safety of autonomous robots has been recognized in the literature and different verification approaches have been proposed.

For instance, [22] proposes a framework for the verification of Golog plans. The methodology is based on model checking and first-order theorem proving. The safety of the plan is verified with respect to temporal properties.

A similar approach is used in [10], where again safety of Golog programs is checked. To guarantee decidability of the verification problem, the input language is restricted.

In [114] the authors manually model the robotic plan in the modelling language Brahms, which is then automatically translated to PROMELA and verified with the SPIN tool. In contrast to the Turing-complete language CPL that we investigate in this thesis, Brahms is limited to if-then-else rules to model the robots behaviour.

In [28] model checking is again used for the verification of the robotic software. Here, the authors check the robots actions against its own belief state. Instead of trying to verify that the robot performs the correct action, they verify that the robot always performs an action that it believes to lead to a safe state.

In [98] the translation of several domain-specific languages into SMV is presented. The authors also restrict themselves to verify the robotic plan with respect to internal properties, but remark that modelling the environment and verifying with respect to environmental properties is "very important, [however] modeling the environment is very complex, and is not at all well understood at this point".

Another approach to plan verification is probabilistic model checking, where probabilities for erroneous behaviour is computed. Examples include [79, 80].

For a broader overview of robot plan verification, see [64].

Overall, most current approaches to robot plan verification consider internal properties of the plan without a model of the robots environment. We would however like to argue that a robot's behaviour can only be properly judged when its effect and interaction with the environment are taken into consideration. In this thesis, we therefore decided to combine formal verification techniques with an explicit model of the environment. We also use

symbolic execution as our verification technique. In contrast to model checking, symbolic execution can act directly on the plan code and requires no further modelling step.

Another approach to ensure the safety of autonomous robots is *runtime verification*, also referred to as *monitoring* [42, 43]. In contrast to formal verification methods, runtime verification is only able to detect errors when (or just before) they occur and is not able to prove the safety of the robotic plan.

3.2 MODELLING OF ROBOTIC ENVIRONMENTS

Logic-based models of robotic environments have a long history. One of the earliest formalisms in this direction is the Situation Calculus [68, 88], which is based on first-order logic. The situation calculus models the state and history of the environment as a *situation*. *Fluents* are then used to express that some property holds in a set of situations. Executing an *action* in some situation S results in a new situation S'. The situation calculus allows the modelling of action preconditions and effects. One also has to model the non-effect of actions, i. e. that an action does not change certain unrelated properties of the world. Due to the situations representing the history of the environment instead of just the current state, a model in the situation calculus represents a tree-like structure of all possible environment states.

The *fluent calculus* [104, 105] is a variation on the situation calculus. Here, the environment state is not described through situations, but rather through *states*. Each state is a concatenation of all fluents that hold in that state.

Another formalism is the *event calculus* [53, 72, 92], which is also based on first-order logic. In contrast to the situation calculus, the state of the environment is not expressed through situations, but rather through timepoints. At each timepoint a subset of the fluents holds, thus describing the state of the environment at that timepoint. Actions in the event calculus have no explicit precondition. Instead, the effect of an action may depend on the state, including actions that have no effect at all under certain preconditions. Due to the semantics based on timepoints, a model in the event calculus represents a linear description of the environment states. In [76], a discrete version of the event calculus has been presented. In Section 4.2 we use this *discrete event calculus* (DEC) and will therefore describe it in more detail there.

Other examples of formalisms for the description of actions and environments are the action languages $\mathcal{A}$ [35], ADL [83] and PDDL [36] and their extensions. In contrast to the calculi described above, these formalisms have a restricted expressive power which allows for efficient reasoning. In many cases, properties can be proven even for an unrestricted time period. On the other hand, this limited expressive power also limits the environments that can be modelled. For instance, non-determinism, ramification constraints, gradual change, or multiple agents can all be expressed in the DEC, but are often problematic for those action languages.

In this thesis we make use of these formalisms, in particular the DEC, to enhance our presented verification technique. To the best of our knowledge, this use of formal environment models and especially the combination with symbolic execution is a novel research direction.

We also present techniques to automatically build and debug formal environment models. The debugging of formal models is still a manual process. Similarly to the software domain, tools that assist in debugging have been developed [66]. 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.

3.3 FUZZING FOR ROBOTICS

Fuzzing has been mostly applied in the security domain, where it is used to generate unexpected inputs that a program is not able to handle properly. The fuzzing process can be unguided or guided by different policies or metrics. In coverage-guided fuzzing, the code coverage is used to find the next input. There are several mature tools for coverage-guided fuzzing such as AFL [120] or libfuzzer [63]. Since many applications require inputs to be in a certain format, a major research direction is the selective generation of valid inputs such as specific file formats [17, 87].

The application of fuzzing to functional safety in the robotics domain is still a new research direction. Nonetheless, there are already some promising applications.

In [26] fuzzing is used to generate inputs to an autonomous robot or its subroutines. The fuzzer is restricted to a certain grammar to provide valid inputs, but is otherwise not guided.

In [117] the fuzzer is used to generate an environment for a robotic agent. The generated environment is however only static, unlike the environments we present in Chapter 6, which also include dynamic, manipulable objects. In addition, the guidance for the fuzzer is based on machine learning instead of the code coverage.

The tool PGFuzz [48] is able to generate inputs to the robot's software. In contrast to this work, the fuzzing is guided by a logic-based policy and the SUT is a lower-level control system instead of a high-level plan.

In summary, fuzzing in the robotic domain is still in its infancy. The existing approaches are not plan-based nor coverage-guided. In addition, most approaches only generate inputs to the control programs methods instead of generating a full environment.

Other coverage-guided methods for autonomous robots have also been proposed. In [9] a tool for automatic test pattern generation (ATPG) based on structural coverage metrics is discussed. In contrast to fuzzing, ATPG will produce less test cases and may therefore miss certain edge cases even though the robotic software has been fully covered by some structural metric.

In [6] the authors present a novel coverage metric, *situation coverage*. The authors argue that structural coverage metrics are not sufficient, since the situation that the robot finds itself in is far more meaningful than the path through the source code. In Chapter 6 we use a similar argument for our novel coverage metric. In contrast to the situation coverage, our presented coverage metric is only dependent on the robot's actions and needs no manual definition of relevant situations.

4

SYMBOLIC VERIFICATION OF ROBOTIC PLANS

The safety and correctness of autonomous robots is vital for their success inside human environments. The current state-of-the-art of manual, simulation-based testing is however insufficient to find some of the more hidden bugs inside the robotic plan. We therefore propose to use formal methods, in particular symbolic execution, to ensure the correctness of robotic plans and uncover bugs. The approaches presented in this chapter have been implemented for the plan language CPL in our tool *Symbolic Execution Engine for Cognition-Enabled Robotics* (SEECER). We present three major contributions in this chapter.

First, in Section 4.1 we introduce SEECER and our general approach to symbolic execution for robotic plans. Here, we also model the robots environment in CPL to enable an easier integration with the plan. We evaluate our approach on an abstract formalized environment.

In Section 4.2 we focus on a better environment modelling. Here, we use the *Discrete Event Calculus* (DEC), a logical formalism widely used in the robotics domain. We integrate modelling and reasoning routines for the DEC into SEECER and evaluate our approach on a realistic household environment.

Section 4.3 focusses on the failure handling mechanisms in CPL. We use symbolic fault injection on top of the symbolic execution to find gaps in the failure handling of robotic plans. We evaluate this approach on a set of generalized fetch and deliver plans.

Finally, Section 4.4 concludes this chapter and gives directions for future work.

4.1 SYMBOLIC EXECUTION OF ROBOTIC PLANS

In this section, we propose the first symbolic approach for verifying plans of cognition-enabled autonomous robots that perform everyday tasks in human environments. We use the plan language CPL and the CLisp bytecode as a basis for our verification. We envision a verification methodology based on symbolic execution, as it has been shown that symbolic execution is a highly effective technique for finding deep errors in complex software applications. We present here the symbolic execution engine SEECER for CPL and the CLisp bytecode. SEECER allows to check plan correctness with respect to environment models as well as annotated assumptions and assertions.

We begin with background information on the formalized environment in Section 4.1.1. Afterwards, Section 4.1.2 presents the main contribution, i. e. the verification approach for CPL. Finally, we show the applicability of our approach in a case study in Section 4.1.3.

4.1.1 *Background: Wumpus World*

Autonomous robotic agents find themselves in highly complex environments, which exceed the limits of exhaustive reasoning. In this section we introduce the *Wumpus World*, a formally defined environment that operates on relatively simple rules, but still poses a challenge due to its incomplete information available to agents. We will use the Wumpus World as a running example and environment for our experimental evaluation.

Definition 5. *The Wumpus World is defined as a rectangular grid of cells to which Cartesian non-negative integer coordinates are assigned. We define* $0, 0$ *to be the cell in the southwest corner. 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. However, 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. In doing so, the agent might face obstacles in terms of the Wumpus*, a dangerous creature emitting a bad odour 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 defence, 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.

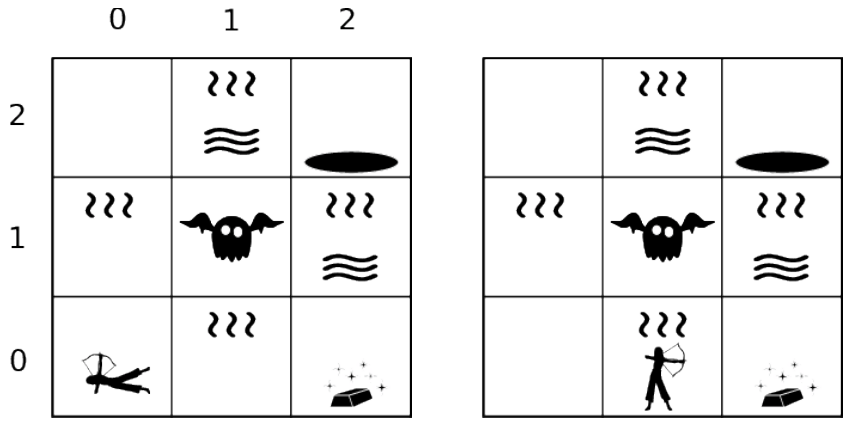
The agent may perceive signals to gain information about the world around him. These are a stench *signal in rooms adjacent to the Wumpus, a* breeze *in rooms adjacent to one or more pits, a* glitter *in the room with the gold or a* bump *when the agent tries to walk through the boundaries of the dungeon.*

To interact with the world, the agent may perform several actions: turning *by 90 degrees in either direction,* walking *one room forward,* grabbing *the gold,* shooting *the arrow,* climbing *out of the dungeon, and* perceiving *any of the signals.*

Example 7. *Consider the* 3×3 *Wumpus World in Fig. 8a. 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* $2, 2$*.*

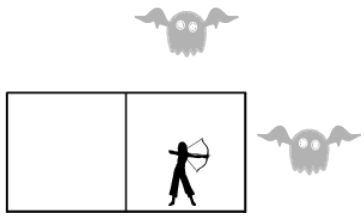
Stenches emitted by the Wumpus are depicted as vertical curvy lines while breezes swirling around pits are horizontal ones. The glittery shine of gold is represented as twinkling stars.

Due to the incomplete information given to the agent, the agent's belief state usually contains only parts of the full environment.



(a) A 3×3 Wumpus World

(b) Environment after one walking action



(c) Agent's belief state

Figure 8: Environment vs. belief state

Example 8. *Assume, the agent in the Wumpus World depicted in Fig. 8a would turn to its right and would walk to room* $1, 0$*. The resulting world is shown in Fig. 8b. A perceiving action for stenches 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. 8c depicts the agent's belief state after the first walking and perceiving action. They know that they walked in eastern direction from their starting position. Since they detected a stench, a Wumpus might hide in either adjacent room.

4.1.2 *Formal Verification of CPL Plans*

In this section, we propose a verification approach for high-level plans for autonomous robotic agents based on symbolic execution. We start with an overview and the general idea in Section 4.1.2.1. We then go into detail about how we deal with the interaction between the plan and its environment via an interface in Section 4.1.2.2. Afterwards, we describe in detail the symbolic execution on the resulting CLisp bytecode plans in Section 4.1.2.3.

4.1.2.1 *Overview*

This section summarizes the approach on CPL verification before we go into more detail in the following sections.

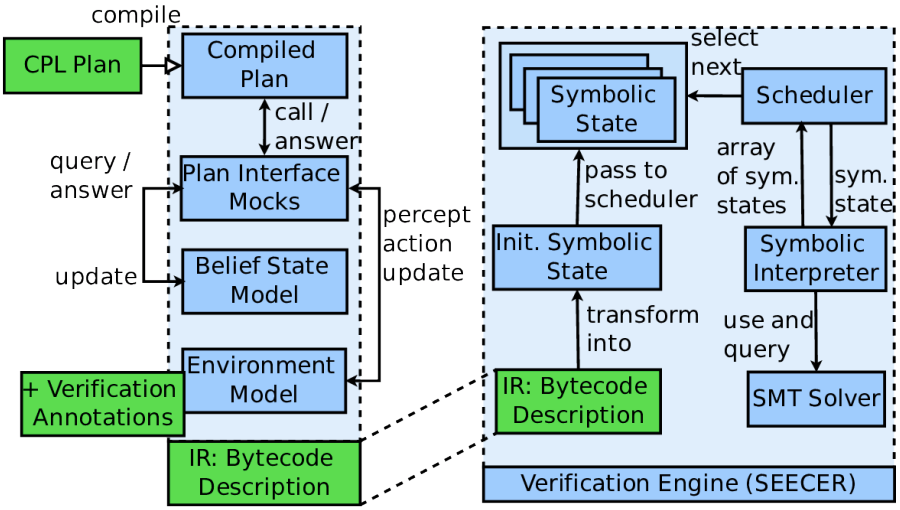


Figure 9: Overview of proposed plan verification approach

Since symbolic execution is simpler when dealing with a linearised source code instead of nested expressions as they occur in Common Lisp, we require a linearised *intermediate representation* (IR) of the plan, environment and belief state of the robot. We choose the CLisp bytecode introduced in Section 2.1.2 as our IR.

Consider Fig. 9 for an overview of our approach. Our goal is to formally verify that certain safety constraints formulated as assertions hold with respect to a given CPL plan.

We start with compiling the CPL plan into CLisp bytecode. Additionally, we integrate an environment model as well as the agent's belief state 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 directly on the environment model and query the belief state model. Note that it is possible to exchange the environment model without modifying the plan and hence, to verify the same plan's safety in different environments.

For verification purposes, symbolic expressions in combination with assumptions and assertions (verification annotations) are embedded into the IR (see lower left green box in Fig. 9). This enables a comprehensive state space exploration.

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

Our contribution includes (1) the integration of mocked functions and verification annotations into CPL and their translation into the CLisp bytecode, and (2) SEECER, which is the symbolic execution engine tailored for the bytecode.

```
1 (perform (an action (type turning) (direction right))
     )
2 (perform (an action (type walking)))
3 (perform (an action (type walking)))
4 (perform (an action (type perceive) (signal glitter))
     )
```

Figure 10: Designators in the Wumpus World

4.1.2.2 *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. The main interface between the CPL plan and the Wumpus World environment is CPL's `perform` function. The following example illustrates how the `perform` function can be used to perform actions in the Wumpus World.

Example 9. *Reconsider the* 3×3 *Wumpus World given in Example 7 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 sequence of* `perform` *calls depicted in Fig. 10. The first call would make it turn to its right (i. e. in eastern direction). The second and third call would make it 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 location.*

Even though `perform` offers an intuitive interface for programmers, the underlying complexity of calls like `perform` in plan languages is non-trivial in domains of symbolic execution. In particular, we want to abstract from the concrete implementation of the underlying perception, navigation and manipulation modules. 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 execution stack. Mocking in this context means creating a function, which to the plan *behaves* like `perform` would do without calling the underlying stack and thus reducing complexity. For discrete and finite worlds such as the Wumpus World, we use the intended behaviour for every possible `perform` call to dynamically create this mock for `perform` 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 plan language. For our ongoing explanation, 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 3. Dependent on that type, actions are performed. In the Wumpus World, these can be of type *turning* (Line 4), *walking* (Line 10), *grabbing* (Line 15), *shooting* (Line 19), *climbing* (Line 24), and *perceiving* (Line 27).

Algorithm 1 CPL `perform` mock

Input: Action designator d

```
1:  if typed ≠ perceiving then
2:      bump ← false
3:  switch typed do
4:      case turning
5:          dir ← directiond
6:          if dir  right then
7:              turn 90° clockwise
8:          else
9:              turn 90° counterclockwise
10:     case walking
11:         if agent faces a wall then
12:             bump ← true
13:         else
14:             go one step in viewing direction
15:     case grabbing
16:         if agent_x  gold_x ∧ agent_y  gold_y then
17:             remove the gold nugget from the world
18:             has_gold ← true
19:     case shooting
20:         if has_arrow then
21:             has_arrow ← false
22:             if Wumpus is located in viewing direction then
23:                 remove Wumpus from world
24:     case climbing
25:         if agent_x  0 ∧ agent_y  0 then
26:             leave dungeon
27:     case perceiving
28:         ...
```

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. This signal can be perceived by the agent to let them know that they walked into a wall. Any action besides perceiving makes the bump signal disappear again (Line 2).

We omit the *perceiving* implementation in the pseudo code (Line 28) due to its large size. 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 modelling of both deterministic and non-deterministic environment models through the use of additional symbolic variables and assumptions inside the environment model.

4.1.2.3 *Symbolic Execution for CPL*

In this section, we present our Symbolic Execution Engine SEECER for the CLisp bytecode that was mentioned over the previous sections. The right part of Fig. 9 on page 26 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. CLisp bytecode 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 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. division by zero.

SEECER starts with a combined CLisp bytecode description which integrates the environment model, the belief state model, and the actual plan. The 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. This is important when 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.

SYMBOLIC EXECUTION STATE A symbolic execution state can be defined as the tuple pc, ip, v, g. The path condition pc describes the preconditions needed to reach the current path. The instruction pointer ip points to the next bytecode instruction to be executed. The value stack v and the global variable map g store the current value of all variables. Variables are stored in the form of *cells*.

A cell can contain any structure that may be formulated in current CPL plans. We support integer values, real values, Booleans, strings, Common Lisp symbols, functions, lists, CLOS classes and CLOS objects, including classes and objects defined by CPL such as designators. Integers, reals, and Booleans may be represented as concrete values or possibly symbolic SMT expressions. Lists, classes and objects are not symbolic, but their contained values may be.

The engine starts with *ip* pointing to the first line of the first top-level function. This corresponds to the entry point of the CPL plan. The path condition *pc* is set to $\top$ and the stack *v* and mapping *g* are empty.

SYMBOLIC INTERPRETER The interpreter executes bytecode instructions one after another and updates the symbolic execution state accordingly. We extended the symbolic interpreter beyond the built-in bytecode functions by including `assume`, `assert`, `sym-int`, `sym-real` and `sym-bool` functions. The `assume` function adds its argument to the current path condition. The `assert` function initiates an SMT call to check if its argument can be violated under the current path condition. If that is the case, an error has been found and SEECER terminates and returns a *counter example*. Based on the counter example, it is possible to retrieve the state of the environment model and CPL plan as well as the assertion that has been violated. The *symbolic instructions* `sym-int`, `sym-real` and `sym-bool` create a new symbolic variable of the respective type, which can now be used. The variable is initially unrestricted.

Every instruction except for control flow instructions increases the *ip* by one. Whenever a new variable is introduced via a Common Lisp instruction or a symbolic instruction, a new cell is added to either *v* or *g*, depending on the instruction's semantics.

For branch instructions like `JMPIF` with condition *c* and target label *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 the target 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), the current state *s* is replaced with two new states $s_\top$ and $s_\bot$, defined as follows:

$pcs_\top$	$pcs \wedge c$	$pcs_\bot$	$pcs \wedge \neg c$
$ips_\top$	l	$ips_\bot$	ips 1
$vs_\top$	vs	$vs_\top$	vs
$gs_\top$	gs	$gs_\top$	gs

Essentially, $s_\top$ continues as if *c* was $\top$ and $s_\bot$ as if *c* was $\bot$. Please note that an SMT solver is only employed if *c* is symbolic and not trivially simplified to $\top$ or $\bot$. Furthermore, only one clone operation is necessary to obtain $s_\top$ and $s_\bot$, because the current state *s* is re-used. The interpreter returns $s_\top$ and $s_\bot$ to the scheduler.

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

4.1.3 *Experimental Evaluation*

We conducted a case study by assembling all individual components described in the previous sections. These include the approach to combine environment models and plans enriched with safety annotations as well as our verification engine SEECER to test those annotations. In this following section, we give an overview of our results.

We have implemented our verification approach for high-level robotic plans as the symbolic execution tool *SEECER* in C++.

As a case study, we consider two CPL plans acting on the Wumpus World. 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 version 4.8.0. We configured Z3 to use its stack-based incremental solver, since it has been shown to be particularly effective for symbolic execution [62]. In the following we describe our two plans (Section 4.1.3.1), the verification annotations (Section 4.1.3.2) and the results of the experimental evaluation (Section 4.1.3.3) in more detail.

4.1.3.1 *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.

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.

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 southern-most room in its column, move one column in eastern direction 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 10. *The function in Fig. 11 is part of the Column-wise Plan. It is supposed to determine if a room's neighbourhood is safe. The result of this function is used to guide the agent's 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)))
```

Figure 11: Function is-neighborhood-safe

The function first checks for a breeze in the current room (Line 2). If it encounters a breeze, the current neighbourhood 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 neighbourhood is labelled as safe iff the stench has vanished (Line 9).

The faulty version of this plan misses the negation in Line 9 of Fig. 11. This will cause the agent to sometimes label an unsafe neighbourhood as safe, which might lead to dangerous exploration.

4.1.3.2 *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 behaviour 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.

Table 1: 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)

4.1.3.3 *Experimental Results*

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 4.1.3.2.

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 found a counterexample demonstrating the bug in the CPL plan leading to unsafe behaviour in less than a second. In most domains, finding bugs is easier than proving their absence. As the following results show, our approach is no exception to this.

Table 1 shows the results for the safe versions of the *Slalom* plan (upper half of Table 1) and *Column-wise* plan (lower half of Table 1). 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 behaviour (i. e. none of the assertion classes specified in Section 4.1.3.2 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 size of the environment model as well as the number of pits 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.

The slalom plan requires a maximum 265 symbolic paths and at most 16 minutes for SEECER to solve its correctness. The column-wise plan on the other hand required over 10000 symbolic paths and reached the time limit of 8 hours four times. Interestingly, the number of pits seemed to have a higher negative impact here than just the size of the world. We expect further optimization techniques to decrease the number of symbolic paths and thus improve the scalability on environments like the Wumpus World.

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 high-level robotic plans and indicates that the general approach can be a suitable foundation to deal with larger and more complex environments and plans.

4.2 VERIFICATION VIA LOGIC-BASED ENVIRONMENT MODELLING

Reasoning about the robotic plan in isolation can only offer limited benefits. In particular, interaction between the robot and its environment needs to be taken into account to achieve meaningful verification results. In the previous section, we presented an approach where the environment was modelled directly in CPL. While this allows for an easier integration in SEECER, the environment has to be written specifically for the plan under verification. Instead, the modelled environments should be reusable not only for the verification of other plans, but also for other reasoning tasks such as planning. In fact, there are several logic formalisms specialized in the modelling of environments and actions, such as the *Situation Calculus* or the *Event Calculus*. They are regularly used to model robotic environments and actions. These formalisms have several advantages over a model in CPL, such as their well-defined semantics and a plethora of environment descriptions and reasoning procedures proposed in the literature.

In this section, we propose a safety verification methodology of robotic plans written in CPL with respect to a logically formalized environment description. Our formalism of choice is the *Discrete Event Calculus* (DEC) due to its high expressiveness and simultaneous decidability. Our contribution in this section is threefold. We first present a decision procedure for the verification of simple branching-free action sequences with respect to a DEC environment model. This procedure serves as a building block for our second and major contribution, namely the verification of more complex robotic

plans through a combination of DEC reasoning and symbolic execution. Our third contribution is the verification of several CPL plans in a household environment and the modelling of that very environment in DEC.

We first introduce some background about the DEC in Section 4.2.1. Afterwards, we introduce our safety verification methodology in Section 4.2.2, Finally, Section 4.2.3 presents our experimental evaluation.

4.2.1 *Background: Discrete Event Calculus*

The *event calculus* [53, 72, 92] is an established formalism to model and reason about events and their consequences. It allows for the modelling of non-determinism, conditional effects of events, state constraints, and gradual change, among others. A domain description modelled in the event calculus follows the *common-sense law of inertia*. Intuitively, this means that the properties of the world do not change over time unless there is an explicit reason for the change. The model designer may however choose to *release* certain properties from this law. Furthermore, the event calculus allows to state that a predicate must be false unless explicitly required to be true. This is known as *default reasoning* and can be used e. g. to limit the occurrences of events. Default reasoning is usually realized through *circumscription* and denoted as $\texttt{CIRC}\phi; P$. Here, all occurrences of predicate P in ϕ are false unless specifically required by ϕ to be true.

The event calculus has been used to model robotic sensors [93], traffic accidents [21], diabetic patients [44] and smart contracts [54].

In [76] a discrete version of the original event calculus has been introduced. This section recaps this *Discrete Event Calculus* (DEC). For simplicity, a version without gradual change axioms is presented.

4.2.1.1 *Overview*

The DEC is based on many-sorted first-order logic with equality, supporting the sorts of *events*, *fluents*, *integers*, *timepoints* and arbitrary user-defined sorts (e. g. for domain objects). Events are occurrences in the modelling domain and can be divided into *actions*, which are deliberately executed by an agent, or *triggered events*, which happen as a result of a change in the world. In this thesis, we will focus mostly on actions and will therefore use event and action interchangeably. There exists no notion of preconditions of an action, i. e., any action may happen in any state. The effects of an action can, however, vary depending on the state of the world. Consequently, the same action could lead to the desired effect, an erroneous effect, or no effect at all depending on the surrounding environment. Fluents describe the state of some property of the world through time. At any given point in time, a fluent may be either *true* or *false*. Timepoints in the DEC as opposed to classical event calculus are bounded to the integer domain. Sorts may be *reified*, i. e. taking other sorts as arguments. Examples of this are the action *goinglocation* or the fluent *isAtobject*, *location*.

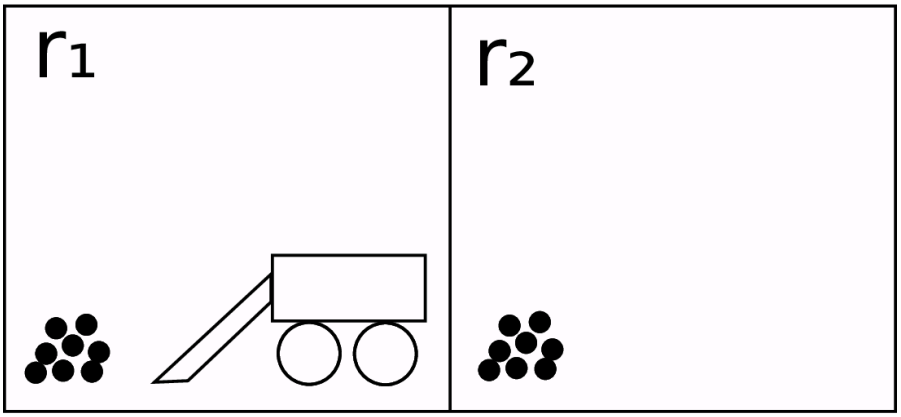


Figure 12: Visualization of the vacuum world (n 2)

DEC descriptions are built using a set of predicates to formalize the state of the world at different timepoints as well as the occurrences and effects of actions. These predicates include:

- *Happensa*, t: Action a happens at timepoint t.
- *HoldsAtf*, t: Fluent f is true at timepoint t.
- *ReleasedAtf*, t: Fluent f is released from the common-sense law of inertia at timepoint t.
- *Initiatesa*, f, t: When action a happens at timepoint t, then fluent f will be true at timepoint t 1.
- *Terminatesa*, f, t: When action a happens at timepoint t, then fluent f will be false at timepoint t 1.
- *Releasesa*, f, t: When action a happens at timepoint t, then fluent f will be released from the common-sense law of inertia at timepoint t 1.
- Arbitrary user-defined predicates.

Additionally, the predicates $\neq, \leq, <, \geq, >$ and the functions $, -, \cdot, \div$ are defined over integers with their usual extensions. To illustrate how these predicates may be used to model robotic environments, consider the following example:

Example 11. *Consider the modelling of a simple robotic environment inspired by the* vacuum world *[90]. The environment is composed of a finite number of rooms $r_1, \ldots, r_n$, which are each either dirty or clean. The rooms are arranged in a row, i. e. room r_i is left of room r_{i1} and right of room r_{i-1}. In the initial state, a vacuum cleaner robot is positioned in one of the rooms. The robot can move through the rooms and clean the room it is currently in. A possible state of the vacuum world with n 2 is visualized in Fig. 12. In this case the robot is located in room r_1 and both rooms are dirty.*

Our DEC description for the vacuum world includes the sort room, which is a sub-sort of the integers, the actions GoLeft, GoRight and CleanRoom and the fluents RobotInRoomroom and Dirtyroom.

At first, we require that the RobotInRoom fluent is functional, i. e. the robot is in exactly one room at any given time:

$$\begin{aligned} &\forall t \ \exists r \ (HoldsAtRobotInRoomr, t) \\ &\forall t, r_i, r_j \ (HoldsAtRobotInRoomr_i, t \wedge \\ &\qquad HoldsAtRobotInRoomr_j, t \ \Rightarrow \ r_i \ \ r_j) \end{aligned}$$

After that we describe the effects of the robot's actions. The GoLeft and GoRight action will move the robot in the respective adjacent room and remove it from its current room, unless it is already in the leftmost (r_1) or rightmost (r_n) room:

$$\begin{aligned} &\forall t, r \ (HoldsAtRobotInRoomr, t \wedge r \neq r_1 \ \Rightarrow \\ &\qquad InitiatesGoLeft, RobotInRoomr - 1, t \wedge \\ &\qquad TerminatesGoLeft, RobotInRoomr, t) \\ &\forall t, r \ (HoldsAtRobotInRoomr, t \wedge r \neq r_n \ \Rightarrow \\ &\qquad InitiatesGoRight, RobotInRoomr \ 1, t \wedge \\ &\qquad TerminatesGoRight, RobotInRoomr, t) \end{aligned}$$

The CleanRoom action will result in the robot's current room being clean (i. e. not dirty):

$$\begin{aligned} &\forall t, r \ (HoldsAtRobotInRoomr, t \ \Rightarrow \\ &TerminatesCleanRoom, Dirtyr, t) \end{aligned}$$

To ensure that these predicates have the intended logical consequences, a set of axioms is necessary. These axioms are given below.

4.2.1.2 *Axioms*

Following the notation from [76], all free variables are assumed to be universally quantified.

Axioms DEC1 through DEC4 deal with gradual change and are therefore omitted here. The axioms DEC5 through DEC8 enforce the common-sense law of inertia, i. e. if a fluent is not released and no action happens to change its value, then the fluent will retain its value from the last timepoint. Additionally, if no action happens to release the fluent, it will remain unreleased. If a fluent is released and no action happens to set it to either truth value, it will remain released.

AXIOM DEC5

$$\begin{aligned} &(HoldsAt f, t \wedge \neg ReleasedAt f, t \ 1 \wedge \\ &\neg \exists a \ (Happensa, t \wedge Terminatesa, f, t)) \ \Rightarrow \\ &HoldsAt f, t \ 1 \end{aligned}$$

AXIOM DEC6
$(\neg HoldsAt f, t \wedge \neg ReleasedAt f, t\ 1 \wedge$
$\neg \exists a\ (Happens a, t \wedge Initiates a, f, t)) \Rightarrow$
$\neg HoldsAt f, t\ 1$

AXIOM DEC7
$(ReleasedAt f, t \wedge$
$\neg \exists a\ (Happens a, t \wedge (Initiates a, f, t \vee Terminates a, f, t))) \Rightarrow$
$ReleasedAt f, t\ 1$

AXIOM DEC8
$(\neg ReleasedAt f, t \wedge$
$\neg \exists a\ (Happens a, t \wedge Releases a, f, t)) \Rightarrow$
$\neg ReleasedAt f, t\ 1$

The axioms DEC9 through DEC12 ensure the correct consequences of actions. That is, if some action happens that initiates (terminates) a fluent, that fluent will be set to true (false) at the next timepoint. The fluent will also no longer be released from the common-sense law of inertia. If some action happens that releases a fluent, that fluent will be released at the next timepoint.

AXIOM DEC9
$(Happens a, t \wedge Initiates a, f, t) \Rightarrow HoldsAt f, t\ 1$

AXIOM DEC10
$(Happens a, t \wedge Terminates a, f, t) \Rightarrow \neg HoldsAt f, t\ 1$

AXIOM DEC11
$(Happens a, t \wedge Releases a, f, t) \Rightarrow ReleasedAt f, t\ 1$

AXIOM DEC12
$(Happens a, t \wedge Initiates a, f, t \vee Terminates a, f, t) \Rightarrow$
$\neg Released f, t\ 1$

Let the conjunction of axioms DEC5 to DEC12 be Ax_{DEC}.

4.2.1.3 *Reasoning*

The following example showcases a possible reasoning problem in the DEC.

Example 12. *Consider again the DEC description from Example 11. We will now use this description to reason about the vacuum world with two rooms (n 2). We require that the robot starts in the left room:*

$$HoldsAtRobotInRoom r_1, 0$$

We additionally specify an action that is executed by the robot:

$$HappensRight, 0$$

When combining this extended description with the Ax_{DEC} axioms, we can infer $HoldsAtRobotInRoom r_2, 1$ as a logical consequence. Please note that this consequence is true and can be deduced even though we did not specify some aspects of the initial state, namely the dirtiness of the rooms.

The former is an example of the *deduction* reasoning task. Deduction asks whether a certain goal state follows from a (partial) initial state and a set of actions. Other notable reasoning problems are *abduction* which asks for a sequence of actions that lead from a given initial state to a given goal state, and *model finding* which asks for complete models of partially specified DEC descriptions.

Since most interesting reasoning tasks in first-order logic are generally undecidable, reasoning in the classical event calculus has to be done either manually [73, 95] or automatically in highly restricted settings [94]. The DEC on the other hand allows for fully automated reasoning by restricting all domains, including the timepoints, to finite sets. We call these descriptions *bounded DEC descriptions*. One way to reason about such bounded DEC descriptions is a translation into *Boolean satisfiability* (SAT). For this purpose, universal (existential) quantifiers are replaced by a conjunction (disjunction) over all objects of the respective sort and the resulting quantifier-free formula is converted into *Conjunctive Normal Form* (CNF). This together with efficient computation of circumscription and simplification techniques was implemented in the Discrete Event Calculus Reasoner (DEC reasoner) [76]. The resulting Boolean formula can then be solved by state-of-the-art SAT solvers, yielding a set of models, which can be translated back into models for the original DEC description.

4.2.2 *DEC-based Verification of Robotic Plans*

In this section, we propose a novel methodology for verification of robotic plans with respect to environment descriptions formalized in DEC. We give an overview of the considered topics and the structure of this part in the following Section 4.2.2.1. In Section 4.2.2.2, we first cover the verification of simple action sequences and in Section 4.2.2.3, we present our verification approach, which is based on symbolic execution, for complex plans written in CPL.

4.2.2.1 *Overview*

Robotic agents operating in complex and changing household environments can impose a safety risk on both the environment and themselves. To verify the safety of plans operating in these environments, we present an approach that combines symbolic execution and DEC reasoning. The problem that we are tackling is depicted in Fig. 13 and intuitively reads as follows: given a robotic plan and a DEC description consisting of an environment description, the DEC axioms and a set of safety properties; is it possible to pick values for the free (input) variables (e. g. the position of certain objects) such that any of the safety properties does not hold? The approach that we are proposing implements the verification engine shown in Fig. 13 via a combination of symbolic execution and DEC reasoning and either returns “Safe”, stating the plan’s safety under all possible free variable assignments, or an execution trace and a sequence of environment states as a counterexample leading to

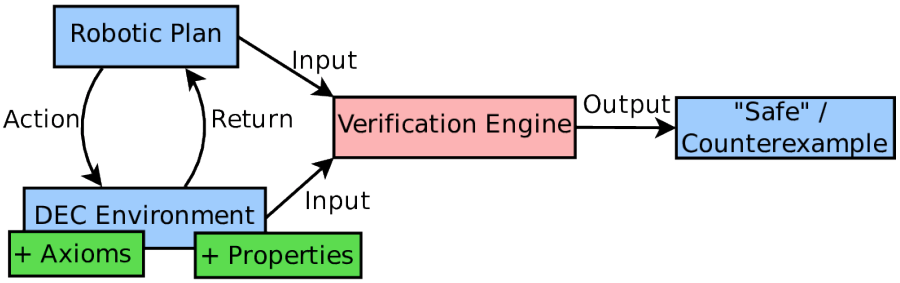


Figure 13: Abstract view on the considered verification problem

the violation of at least one property. An important building block of our approach is a procedure for the verification of action sequences, i. e. a finite, branching-free sequence of atomic actions that are executed in order by a robotic agent. This building block is implemented by means of a reduction to a pure DEC reasoning problem. Since action sequences are still widely used e. g. in manufacturing tasks, it is also useful as a stand-alone technique. In the overall approach for the verification of CPL plans, this procedure is used repeatedly during symbolic execution. We first introduce the verification approach for action sequences in the following section and, afterwards, present our combined approach for complex plans.

4.2.2.2 *Verification of Action Sequences*

Verification of action sequences can be reduced to a pure DEC deduction problem, as we will show in the following. Given the DEC axiomatisation $\mathrm{Ax}_{\mathrm{DEC}}$, an environment description *Env*, a sequence of actions $a_1, \ldots, a_k$ and a set of properties $P_1, \ldots, P_l$, we want to prove that the conjunction of DEC axioms, environment description and action occurrences entails the safety properties, i. e.

$$\mathrm{Ax}_{\mathrm{DEC}} \wedge \mathit{Env} \wedge \mathtt{CIRC} \bigwedge_{i1}^{k} \mathit{Happensa}_i, i-1; \mathit{Happens} \mid \bigwedge_{j1}^{l} P_j.$$

Here, CIRC is the circumscription operator introduced in Section 4.2.1. In this case, it ensures that no actions other than $a_1, \ldots, a_k$ are occurring.

Since most reasoners for DEC, including the DEC reasoner introduced in Section 4.2.1, do not directly support deduction, we formulate the deduction problem given above as a model finding problem instead. To this end, we perform model finding on the following conjunction

$$\mathrm{Ax}_{\mathrm{DEC}} \wedge \mathit{Env} \wedge \mathtt{CIRC} \bigwedge_{i1}^{k} \mathit{Happensa}_i, i-1; \mathit{Happens} \wedge$$
$$\mathtt{CIRC} \bigwedge_{j1}^{l} \neg P_j \Rightarrow U; U \wedge U,$$

where U (short for *unsafe*) is a new 0-ary predicate symbol. Intuitively, we enforce that any violated property sets the predicate U to true and then try to find models that have at least one violated property.

Since the final action occurs at timepoint $k-1$, it is sufficient to consider the timepoints 0 to k. This allows to encode the verification problem in a bounded DEC description and to solve it using the SAT-based DEC reasoner from [76]. If a model is found, it contains concrete states for all timepoints together with the failed properties. This can be helpful when debugging the action sequence. If no model is found, the action sequence is proven to be safe.

Example 13. *Consider again the vacuum world with n 2 from the previous examples. Consider further the following action sequence: GoLeft, CleanRoom, GoRight, CleanRoom. Assume that we want to verify that this action sequence results in all rooms being cleaned after the last action, i. e. at timepoint 4. We express this by the property P_1 $\forall r$ $(\neg HoldsAtDirtyr, 4)$. The verification is now conducted by model finding on the following conjunction:*

$$Ax_{\text{DEC}} \wedge Vac_2 \wedge \texttt{CIRC}HappensLeft, 0 \wedge HappensClean, 1 \wedge$$
$$HappensRight, 2 \wedge HappensClean, 3; Happens \wedge$$
$$\texttt{CIRC}(\exists r \ HoldsAtDirtyr, 4) \Rightarrow U; U \wedge U,$$

where Vac_2 is the DEC description of the vacuum world described in Example 11 with n 2. When giving this conjunction to the DEC reasoner, no model will be returned, therefore proving the safety of the action sequence with respect to P_1 for all possible initial states of the vacuum world.

4.2.2.3 *Verification of Complex Robotic Plans*

In the previous section, we discussed how simple action sequences can be verified with respect to a set of properties using DEC reasoning. This approach is however no longer sufficient to solve the verification task for arbitrary plans written in Turing-complete plan languages. In this section, we therefore combine this procedure with symbolic execution. We present our approach utilizing CPL as a running example. We would like to point out, however, that our approach works for any robotic plan language, as long as a suitable symbolic execution engine is available.

Fig. 14 shows an overview of our architecture. The inputs to the verification problem are a CPL plan and the DEC environment description that interacts with the plan through actions and their respective return values. The environment description is further extended with the DEC axioms and the safety properties, forming a single joint DEC description. The core of our approach is the symbolic execution engine *DEC-SEECER*, which is an extension of the CRAM symbolic execution engine SEECER. We extended SEECER by the capability to handle DEC descriptions and to reason about them in combination with the SMT constraints for the path condition that arise during symbolic execution. An important part of this extension is the interface to the DEC reasoner, which receives DEC descriptions and translates them into Boolean CNF formulae. These formulae can be combined with other SMT constraints and solved by the SMT solver *Z3*. Again, a `perform` mock is used to abstract from low-level effects like motor control. However, this mock is

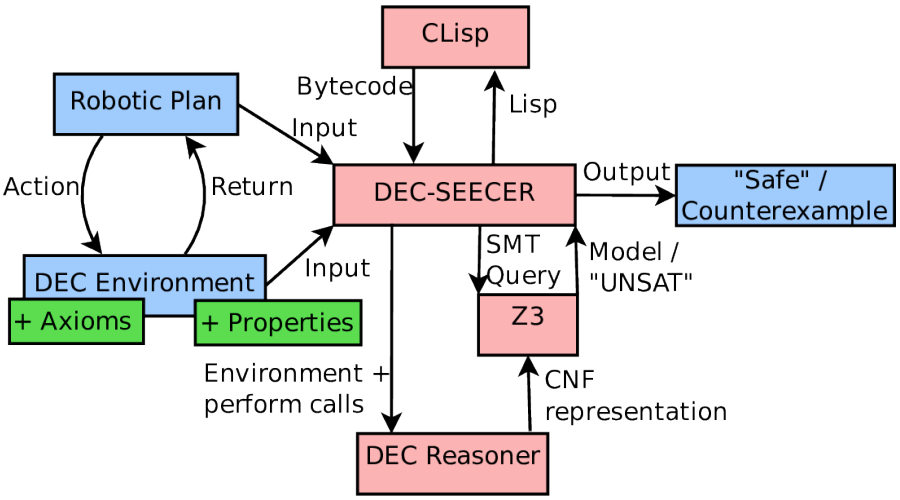


Figure 14: DEC-centric architectural view

not hand-written for each environment, but can instead handle arbitrary DEC descriptions, thus supporting a multitude of different environments.

In the remainder of this section, we describe DEC-SEECER and especially the integration between symbolic execution and DEC reasoning in more detail.

INTEGRATION BETWEEN DEC AND SYMBOLIC EXECUTION The CLISP bytecode is executed symbolically by DEC-SEECER. The general symbolic execution operates similar to the version described in Section 4.1 by managing several execution states. These execution states are however represented by a 5-tuple pc, ip, v, g, E, where pc, ip, v and g are the path condition, instruction pointer, value stack and variable map, respectively. The DEC description E is added to allow combined reasoning about the plan and its environment. This description is built in a very similar way to the one in Section 4.2.2.2. The DEC description of the initial state is given as

$$E_0 \;\; \mathrm{Ax}_{\mathrm{Dec}} \wedge \mathit{Env} \wedge \mathtt{CIRC} \bigwedge_{j1}^{l} \neg P_j \;\Rightarrow\; U; U$$

and combines the $\mathrm{Ax}_{\mathrm{Dec}}$ axioms, environment description and safety properties.

During the symbolic execution of the plan, we differentiate between three types of instructions: the first type are non-control flow Common Lisp instructions, e. g., arithmetic instructions, or string manipulations. These update the execution state in the usual way and do not affect E. The second type are `perform` instructions, which add an action occurrence to E via a respective *Happens* conjunct. Like in Section 4.2.2.2, these *Happens* conjuncts are subject to circumscription. `Perform` instructions also increase the instruction pointer ip by 1, and push a return value onto v. The third type, branching instructions, lead to a feasibility check of both branches. To account for effects

```
1 (perform (an action (type go-left)))
2 (let ((dirty (perform (an action (type detect)))))
3   (if dirty
4     (perform (an action (type clean-room)))))
5 (perform (an action (type go-right)))
6 (let ((dirty (perform (an action (type detect)))))
7   (if dirty
8     (perform (an action (type clean-room)))))
```

Figure 15: CPL plan for the vacuum world

from the environment, the DEC description is incorporated in this feasibility check as follows.

E is translated into CNF by the DEC reasoner. We denote this translation by $DECRE$. Since the SAT variables in this CNF are disjunct from the plan variables, they need to be related via a mapping. This mapping is implemented by the conjunction of equivalence constraints mE. DEC-SEECER now evaluates the satisfiability of both $C \wedge pc \wedge DECRE \wedge mE$ and $\neg C \wedge pc \wedge DECRE \wedge mE$. Here, C and pc are the branching condition and path condition, as before.

To ensure the plan's safety concerning the properties, a similar satisfiability check is used. After executing any action, the following conjunction is checked for satisfiability:

$$pc \wedge DECRE \wedge U \wedge mE$$

Any assignment satisfying this formula corresponds to a counterexample, i. e. an instance of a safety property being violated by the plan. Consequently, if all such checks return *UNSAT* during the symbolic execution, the plan's safety is proven. The following example illustrates our approach.

Example 14. *Consider once again the vacuum world from the previous examples. We extend this world by an additional action Detect that is supposed to detect dirt in the robot's current room. Since this action returns information to the plan, we need an additional fluent ReturnVal(). We also add constraints expressing that Detect will set ReturnVal() to true if the robot's current room is dirty, and to false otherwise. We denote this extended environment description by Vac′.*

Assume we want to verify the safety of the CPL plan shown in Fig. 15. This plan is more complex than the action sequence presented in Example 13 because it considers the state of the environment in Line 3 and 7 before executing certain actions. Namely, the robot only cleans a room if it detects dirt in that room. Again, we would like to verify the plan's safety using the property P_1 from Example 13. Additionally, we would like to prove that the robot will never attempt to clean an already cleaned room. This is expressed by the safety property

$$P_2 \;\; \forall t, r \;\; (\neg HoldsAtDirtyr, t \wedge \\ HoldsAtRobotInRoomr, t \;\Rightarrow\; \neg HappensCleanRoom, t).$$

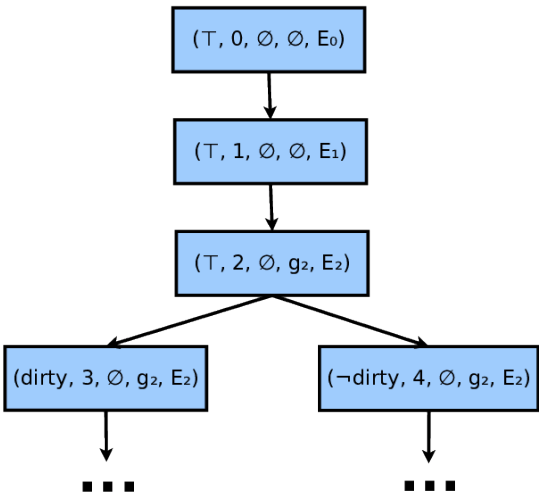


Figure 16: Execution tree of the symbolic execution

The initial symbolic execution state can now be written as the 5-tuple $\top, 0, \emptyset, \emptyset, E_0$ *with*

$$\begin{aligned} E_0 \;\; & Ax_{\text{DEC}} \wedge Vac'_2 \wedge \texttt{CIRC}(\exists r \;\; HoldsAtDirtyr, t_{max}) \;\Rightarrow\; U \\ & \wedge (\exists t, r \;\; \neg HoldsAtDirtyr, t \wedge HoldsAtRobotInRoomr, t \\ & \wedge HappensCleanRoom, t) \;\Rightarrow\; U; U. \end{aligned}$$

Fig. 16 shows parts of the execution tree imposed by the symbolic execution. Each node in the tree represents an execution state composed of the path condition, the instruction pointer (denoted by the respective line number in Fig. 15), variable stack, variable mapping, and DEC description. Since each instruction except for the conditional branch performs an action, the DEC descriptions and assignments are updated as follows:

$$\begin{aligned} E_1 \;\; & E_0 \wedge \texttt{CIRC}HappensLeft, 0; Happens \\ E_2 \;\; & E_0 \wedge \texttt{CIRC}HappensLeft, 0 \wedge HappensDetect, 1; Happens \\ g_2 \;\; & \{\texttt{dirty} \mapsto DECRHoldsAtReturnVal, 2\} \end{aligned}$$

After every action being performed, the plans' safety is checked via an SMT solver call. For example, after the Clean action (which is performed in the node on the bottom left), the following conjunction is checked for satisfiability:

$$\begin{aligned} & g_2 dirty \wedge DECRE_2 \wedge DECRU \\ & DECRHoldsAtReturnVal, 2 \wedge DECRE_2 \wedge DECRU \end{aligned}$$

This formula is unsatisfiable. In fact, every such formula during the symbolic execution of this plan is unsatisfiable, thus proving the safety of the plan.

Since verification of Turing-complete programs is undecidable in general, there are cases in which our approach will not terminate or terminate with an inconclusive result. In particular, this is caused by non-terminating CRAM

plans or complex arithmetic conditions in the plan. These results are exclusively due to the symbolic execution part of our approach, since DEC-based environment descriptions can always be grounded to pure Boolean SAT problems. Because of the undecidability of program verification, termination could only be guaranteed by severely restricting the robotic plans under verification.

In the following section, we show that our approach can nonetheless handle many practically relevant robotic plans.

4.2.3 *Experimental Evaluation*

We implemented DEC-SEECER using the DEC reasoner version 1.0 and the Z3 SMT solver version 4.8.4 as back-end. All experiments have been conducted on a Linux machine with an Intel CPU with 3.5 GHz clock rate. To evaluate our approach, we used several variations of the *Shopping demo* plan taken from the official CRAM repository. The Shopping demo plan involves a two-armed human-sized robot operating in a supermarket environment consisting of a shelf and a table. The robot is supposed to move several objects between the two locations. The shelf is wider than the robot's reach, making it necessary for the robot to determine a suitable position in front of the shelf for grasping certain objects. However, positions directly in front of the shelf cannot be used for detection, because parts of the shelf may obstruct the robot's view. It is, therefore, necessary that the robot first obtains an overview from a suitable position. We modelled these restrictions in an environment description in DEC.

To evaluate our approach, we used several variations of the existing Shopping demo plan in combination with the DEC description. In the following Section 4.2.3.1, we discuss all plans in detail. In Section 4.2.3.2, we present our DEC environment and the safety properties. Finally, in Section 4.2.3.3, we show our experimental results and discuss them.

4.2.3.1 *Robotic Plans*

For the experimental evaluation, a total of six plans have been evaluated. They are listed below.

SHOPPING The original Shopping demo plan attempts to move a set of predefined objects from the shelf to the table. The robot moves to a predefined position from where it has an overview of the whole shelf and tries to detect all objects. Afterwards, it repeats the following operations for each object. First, the robot moves to a central position in front of the shelf. If the object is already within reach, it is then grasped with the closest gripper. Otherwise, the robot needs to move to a different position to its left or right. Once the object is grasped, it is transported to the table and placed onto the tabletop. Over the course of the execution, certain positions on the table are filling up. To avoid collisions, the robot computes a new free position after setting an object and uses that position for the next object.

```
1  (when (>= (y ?object-position) 0.8)
2          (setf ?grasping-arm :left)
3          (perform (an action
4                  (type going)
5                  (target (a location
6                          (pose ?grasp-pose-left))))))
7  (when (< (y ?object-position) -0.8)
8          (setf ?grasping-arm :right)
9          (perform (an action
10                 (type going)
11                 (target (a location
12                         (pose ?grasp-pose-right))))))
13 (perform (an action
14                 (type picking-up)
15                 (arm ?grasping-arm)
16                 (object ?newobject)))
```

Figure 17: Excerpt of the Shopping plan

Fig. 17 shows an excerpt of the Shopping plan. The plan compares an object's position with predefined boundaries (Line 1 and Line 7). Depending on that position, the robot either moves to the left (Lines 2-6), to the right (Lines 9-12), or stays in its current position. Afterwards, the robot attempts to grasp the object (Lines 13-16).

SHOPPING 2 This plan is a modified version of the Shopping plan. A small error was deliberately inserted to test our approach's bug-finding capabilities. By replacing the >= in Line 1 of Fig. 17 with a <=, the robot chooses the wrong grasping position for some objects. We expect this change to result in an error for some initial environment states.

SHOPPING 3 This plan is another erroneous modification of the original Shopping plan. Here, the plan does not move the robot to the designated detection pose at the start of the plan but instead attempts to detect all objects from the robot's initial position. We expect this to result in some objects not being detected, which would mean that some objects remain on the shelf not fulfilling the plan's goal.

SHELF FILLING The *Shelf Filling* plan has the reverse goal of the Shopping demo. A set of objects is located anywhere in the environment and the robot's goal is to pick up these objects and put them onto the shelf. This plan simulates the automatic refilling of supermarket shelves by a robotic agent. Here, each object has an associated row, onto which it has to be placed on the shelf. The plan achieves this by grabbing the objects one by one and placing them in an unoccupied spot in their respective shelf. To this end, it needs to maintain a belief state of objects that have already been placed onto the shelf. This procedure is repeated until there are no more objects left. In some cases,

however, it is necessary to omit certain objects, because some positions on the shelf are initially occupied. Placing these objects is tried again at the end of the plan. This plan is deliberately more complex with a higher amount of branching logic compared to the Shopping plan.

SHELF FILLING 2 We again constructed erroneous versions of the original plan. In this version, whenever an object is omitted, it is simply removed from the list of objects and not moved to the end. We expect this error to result in objects being left in the environment and, therefore, in the wrong position after the plan's termination.

SHELF FILLING 3 This modified version of the Shelf Filling plan does not take certain occupied positions on the shelf into account, resulting in possible collisions of objects.

4.2.3.2 *Environment Description and Safety Properties*

All plans presented in the previous section operate in the same environment consisting of a shelf and a table. We modelled this environment in the DEC. The shelf consists of three rows (top, middle, bottom) and four sections in each row (far left, left, right, far right). Objects may be located in any of the sections in any row, resulting in a total of twelve positions on the shelf per object. There are three positions for the robot in front of the shelf and a fourth one a little further away. These positions are suitable for reaching parts of the shelf or detecting objects on the shelf, respectively. The table is also partitioned into several sections. This allows us to model the limited space available. The table can again be reached from a dedicated position in front of it. Our model uses sorts for the movable objects in the world, the positions, and other aspects like the robot's arms or different heights. We use several fluents modelling the positions of objects and the robot, grasps, detection status, and others. The whole environment model consists of 56 logical sentences.

To ensure the plan's safety, a set of safety properties was also formalized in DEC. These safety properties ensure that (1) the robot never reaches an internal error state, (2) all actions produce their desired effects[1], and (3) no two objects are ever placed in the same position. This last property detects possible collisions that, in the real world, would result in the robot damaging its environment. Additionally, we added properties that require (1) that at the end of the Shopping plan, all objects are placed on the table, and (2) that at the end of the Shelf Filling plan, all objects are placed onto their associated shelf rows.

1 Note that this does not necessarily hold by design of the environment model. E. g. a grasping action will not result in the desired result if the robot is too far away from the object or the gripper is already occupied.

Table 2: Verification results

Plan's name	#LOC	Verdict	#Paths	Time (s)	Time gen. (s)
Shopping	338	Safe	16	2144	1967
Shopping 2	338	Unsafe	2	343	300
Shopping 3	327	Unsafe	1	176	152
Shelf Filling	914	Safe	123	31370	30708
Shelf Filling 2	823	Unsafe	10	2823	2767
Shelf Filling 3	911	Unsafe	11	3326	3262

4.2.3.3 *Experimental Results*

We ran our proposed verification approach on all plans presented in Section 4.2.3.1. All Shopping plans had two objects in their initial state. The objects' positions were not restricted which means that the plan was verified for any possible initial placement of objects. The initial state for the Shelf Filling plans includes three objects. Their positions, both on the table and on the shelf, were again left fully symbolic. In all scenarios, the robot's initial position, arm positioning and torso height was left symbolic to account for all possible starting states.

Table 2 summarizes our experimental results. Here, each row represents a run of one plan. We report (from left to right) the plan's name, the number of lines of the respective CLisp bytecode (#LOC), the verification verdict, the number of paths in the symbolic execution tree (#Paths), the total runtime, and the time spent on generating SAT instances by the DEC reasoner. All times are reported in seconds.

As can be seen, our approach always returned the expected verification result. All errors in the modified plans were found and both unmodified plans were proven to be safe with respect to the specified safety properties. Moreover, the three versions of the Shopping plan were verified with only a few paths and in less than 40 minutes. This is due to the fact that only the branching logic in the plan itself affected the number of symbolic execution paths. Any conditional construct in the environment itself was instead translated into a conditional CNF representation and solved by the SMT solver. The Shelf Filling plans, which were designed to involve a lot more branching, led to more symbolic execution paths and thus to a significantly higher runtime. Even the unmodified Shelf Filling plan was, however, verified in under 9 hours. Verifying the modified versions of both plans took a fraction of the runtime of their unmodified counterparts. This is because DEC-SEECER terminates after the first property violation has been found. The right-most column reports the runtime that was spent on the generation of the SAT instance by the DEC reasoner. As one can see, this procedure was responsible for the majority of the overall runtime (86-98%). The solving process was a lot faster in comparison. This indicates that the generation procedure of

the DEC reasoner is inefficient compared to the solving capabilities of modern state-of-the-art engines. In fact, a number of more efficient grounding procedures have been developed since then (c. f. [46] for an overview). Furthermore, the DEC reasoner does not generate CNF instances iteratively.

In summary, our experiments show DEC-SEECER's capability to verify the safety of robotic plans such as the Shopping demo. Even a more complex plan, namely the Shelf Filling plan, was verified correctly and within an adequate time. To further improve our approach's runtime, a dedicated reasoner for DEC could be developed with state-of-the-art grounding techniques and support for the incremental unrolling of environments.

4.3 SYMBOLIC FAULT INJECTION FOR ROBOTIC PLANS

The dynamic and uncertain environments of autonomous robots pose a significant challenge for many low-level actions, such as grasping an object or navigating to an exact position. While progress is being made in the accuracy of these low-level actions, avoiding failures altogether seems hardly possible. In an autonomous setting, the robot has to recover from low-level failures by itself to be able to still reach its goal. This need for autonomous failure handling has been recognized in the literature and a multitude of failure handling strategies have been described [47, 60, 78]. In plan languages such as CPL, failures may be thrown by low-level modules and handled by higher-level plans. If a failure occurs without an adequate handler, the robotic plan will usually crash, stopping the robot entirely. Ideally, a robotic plan would be written in such a way that all possible low-level failures are handled inside the plan, without any crashes and the need of external interference. With the increasing complexity of robotic plans however, finding unhandled failures is a challenging task. The typical method of finding unhandled failures are simulations. These are, however, inherently incomplete and will often not find failures that occur only occasionally. To tackle this problem, we instead propose to use formal methods, in particular symbolic fault injection to find cases in which failures are not properly handled. Our method is based on the worst-case assumption that any low-level action may fail at any time with any of its possible failure types. We then use our symbolic execution engine SEECER to find all cases in which failures are left unhandled. In this section, we extend SEECER to reason about CPL's failure handling mechanism. We also present a general methodology to implement our worst-case-assumption directly in CPL and present an optimization technique to increase the scalability of our approach. Our method is complete, i. e. it is able to produce either a complete list of all unhandled failures or guarantee that no such failure exists. Since we reason directly on the plan code, we are not limited to certain failure handling strategies.

We present our approach in Section 4.3.1 and an experimental evaluation in Section 4.3.2.

```
(with-failure-handling
      ((failure-type (e)
         handler))
    body)
```

(a) Original code

```
(defun (e) handler1
     (if (typep (type-of e) failure-type)
        handler
        (rethrow-failure e)))
(start-failure-handling "HANDLER1")
body
(end-failure-handling)
```

(b) Rewritten code

Figure 18: Rewriting scheme for failure handlers

4.3.1 *Symbolic Fault Injection for CPL*

This section describes our approach on fault injection for CPL plans.

Our approach takes a CPL plan and tries to find unhandled failures that can occur in it. Most failures are thrown inside very low-level modules that are responsible for the execution of mechanical actions such as grasping. We want to abstract from the internals of these modules, since they often depend on the current state of the environment. Instead, we consider all types of failures that may be thrown by the module and replace the concrete implementation of the module with the worst-case assumption that any of those failure types may occur whenever the module is called.

We call the actions for which we apply this assumption *atomic actions*. Please note that the notion of atomic actions is flexible, i. e. depending on the desired accuracy and runtime, a user could choose higher or lower cut-off points.

The core of our approach is the symbolic execution engine SEECER. We extend SEECER to support the extensive failure handling capabilities that are present in CPL. The details of this extension are given in Section 4.3.1.1.

Our fault injection approach is based on the general worst-case assumption that any action may fail at any time with any of its possible failure types. To reflect this behaviour, we built a general environment model that implements this assumption for all atomic actions of CPL. In addition, we implemented a similar worst-case assumption for all reasoning subroutines. The details of this environment model and the reasoning subroutines are presented in Section 4.3.1.2.

Finally, we propose an additional optimization technique to reduce the runtime of our approach in Section 4.3.1.3.

4.3.1.1 *Extending SEECER with Failure Handling*

Prior to the implementation of this section's functionality, SEECER supported the core of CPL as well as a multitude of Common Lisp functions. To enable symbolic fault injection, the failure handling functionalities of CPL were implemented into SEECER.

The nested Common Lisp code is generally problematic for a symbolic execution engine. Therefore, as a first step, the nested failure handling macro is rewritten into a more sequential control flow. The applied rewriting is illustrated in Fig. 18. Fig. 18a shows the code before and Fig. 18b after our rewriting scheme is applied. As shown, the failure handler is placed inside a new function and guarded by the condition that the type of the active failure is equal to or inherited from the *failure-type* (Line 2 in Fig. 18b). Otherwise, the handler is not executed, but instead the failure is re-thrown to be handled by a higher-level handler (Line 4). We call this new function the *handler function* to differentiate the whole function from the original handler that is now part of it.

The body is encapsulated inside two new functions *start-failure-handling* (Line 5) and *end-failure-handling* (Line 7). SEECER uses these functions to manage failures and handler functions internally. Information about handler functions are organized as a stack, with start-failure-handling pushing a new handler function onto the stack and end-failure-handling removing the top function. When a failure occurs, the execution jumps to the newest handler function on the stack. This is done by pushing a new element onto the function stack similar to a normal function call. The handler function is then executed. If the same or a new failure is thrown inside the handler function itself, the next handler from the stack is executed. If no failure is thrown inside the handler, the failure has been successfully handled and the execution jumps back to the return address, i. e. the line after which the failure was initially thrown. If a failure is thrown with an empty handler stack, the failure reaches the top level of execution. We call these failures *top-level failures*. In a normal execution they lead to a crash of the plan. SEECER can be configured to either terminate the whole execution or just the current context in this situation. In the first case only the first top-level failure would be reported and in the second case all top-level failures would be collected before SEECER terminates. The user can decide between the two modes depending on their needs. If no top-level failure occurs at all, the plan's failure handling is proven to be complete.

4.3.1.2 *Symbolic Substitution of Atomic Actions*

In this section, we want to use the failure handling described in the previous section to find all possible top-level failures. This is based on the worst case assumption that any atomic action may fail at any time with any of its possible failures.

The worst-case assumption is completely implemented into Common Lisp as follows. For each execution of an action, a new symbolic integer is created.

```
1 (let* ((sym-ctr (sym-int symbolic-name)))
2     (if (= 1 sym-ctr)
3         (cram-failure (make-instance
4           'gripper-goal-not-reached))
5         (when (= 2 sym-ctr)
6             (cram-failure (make-instance
7               'gripper-closed-completely)))))
```

Figure 19: Implementation of the worst case assumption for the grasping action

Each of the n possible failures is then assigned a unique integer between 1 and n. A chain of `if` statements now ensures that each failure is thrown iff the symbolic integer has that failure's respective value. For all other values no failure is thrown.

Example 15. *Consider an action of type* Grasping *that tries to grasp an object. There are two possible types of failure that may occur. If the robot's motion planning module does not find a sequence of motions to reach the object – usually because it is too far away – a failure of type* `gripper-goal-not-reached` *is thrown. If the robot instead tries to grasp the object, but detects that its grippers have fully closed (and therefore failed to grasp the object), a failure of type* `gripper-closed-completely` *is thrown. Fig. 19 shows the implementation for the Grasping action. In Line 1 a new symbolic integer is created. The variable* `symbolic-name` *contains a string for the internal naming of the variable in the SMT solver. The symbol* `sym-ctr` *now contains the variable itself. If the variable is equal to* 1*, a* `gripper-goal-not-reached` *failure is created and thrown in Line 4. Similarly, if it is equal to* 2*, Line 7 throws a* `gripper-closed-completely` *failure. For all other values, no failure is thrown.*

Please note that the assignment of integer values to failures is arbitrary. A different assignment will still produce the same outcome as long as all failures are assigned to at least one unique value.

In addition to atomic actions, the environment model also needs to deal with complex reasoning subroutines that are part of many high-level planning frameworks like CRAM. These routines are often complex and take into account the current environment, the robots belief state and dynamically changing knowledge bases. Following our worst-case assumption, we have to reason about all possible results from the reasoning subroutines. Similarly to the atomic actions this is realized through a pure Common Lisp implementation. For each call of a reasoning subroutine, a symbolic variable of the respective type is created. The value may be restricted through `assume` statements (e. g. to a certain numerical range) and is then returned.

4.3.1.3 *State Caching*

Such a general methodology as presented here will often create a large number of symbolic values and result in a large number of symbolic paths. This is

```
1  (defun handler1 (failure)
2      (if (typep (type-of failure)
3                 'gripper-goal-not-reached)
4          (print "WARNING: Grasping failed")
5          (rethrow-failure failure)))
6  (start-failure-handling "HANDLER1")
7  (perform
8      (an action
9          (type grasping)
10         (object obj)))
11 (end-failure-handling)
```

Figure 20: A failure handler without side effects

due to the branching that takes place whenever an action is executed. Therefore, this section discusses a technique to reduce the search space through means of *state caching*.

In many cases a failure handler will have little to no side effects, i. e. the state after the handler has finished is identical to the state in which no failure occurred in the first place.

Example 16. *Consider the plan excerpt in Fig. 20. As shown in the previous example, there are three possible outcomes of the grasping action performed in Lines 7-10: The action may throw a* `gripper-goal-not-reached` *failure, a* `gripper-closed-completely` *failure or no failure at all. In the first case, the handler would print a warning and then jump back to Line 11 and in the third case the handler would not be called at all and instead the execution would continue as normal, also with Line 11. Since the handler has no side effects at all, both execution states would be identical.*

Since both states are identical, they will lead to the same execution traces upon further symbolic execution. Because of this, it is safely possible to only continue execution on one of the states without affecting the final result. This is an instance of *state merging* [55], an established optimization technique in symbolic execution. Usually, state merging will compare symbolic states and if two states are similar by some metric, their path conditions and variable assignments will be combined. This combination results in less, but more complex symbolic states, shifting complexity from the search algorithm to the SMT solver. For an effective use of state merging, a large number of symbolic states must be active at the same time. As demonstrated above, states during fault injection for typical CRAM plans are not only similar, but identical. This enables us to use a simpler version of state merging with less overhead which we call *state caching*. State caching only acts on identical states at certain manually chosen checkpoints in the plan. At the checkpoints the states are stored inside a cache. Whenever a state s is identical to a previously stored state s', the current execution trace can be terminated, since all results after state s have already been produced after s'. This way identical states are only processed once.

The comparison between states is based on the states' function stack, value stack, variable assignments and path condition. These also include values that have been used prior to the current point in the execution, but are not relevant to any decisions after that point. This is especially true for the symbolic values introduced in the last section as they are used only once immediately after being created. These values can therefore be ignored when it comes to comparing two states.

Example 17. *Consider again the previous two examples. Since the symbolic variable* `sym-ctr` *is used to differentiate between the type of failure that is thrown, the two states from the previous example will have differing path conditions. The state which throws the* `gripper-goal-not-reached` *failure, will have* `symctr` 1 *in its path condition. The state which did not throw a failure will instead have a path condition of* `sym-ctr` $\neq 1 \wedge$ `sym-ctr` $\neq 2$. *Since* `sym-ctr` *is not used in any future paths of the program, this difference may however be ignored, making the two states identical again.*

Currently the values to ignore are determined manually, but for future work we plan to do this automatically based on the plan's control flow.

We implemented the state cache in a map structure using hash values for fast comparisons. These hash values are constructed by first hashing the individual entries in the function stack, value stack, variable assignment and path condition and then combining all entries via the XOR function.

4.3.2 *Experimental Evaluation*

This section presents our experimental evaluation. All experiments were conducted on a Linux machine running an Intel CPU with 2.50 GHz clock rate. Our main research questions are whether symbolic fault injection is suited for plan-based robotics and how fast our approach is for typical CRAM plans. We investigate a system of CRAM plans for generalized fetch and deliver actions. These plans are described in more detail in Section 4.3.2.1. The final results and their interpretation with regard to our research questions are presented in Section 4.3.2.2.

4.3.2.1 *Robotic Plans and Actions*

We evaluate our approach on a system of generalized fetch and deliver plans. The plans implement different subroutines that are used to transport objects from one place to another, including searching for objects and opening and closing containers. The plans are very general, i. e. they are independent of concrete objects or locations. They can be roughly grouped into three classes: atomic actions (e. g. `setting-gripper`), low-level plans (e. g. `picking-up`) and high-level plans (e. g. `fetching`). Here the low-level plans use atomic actions internally and high-level plans use atomic actions and low-level plans. One high-level plan, *Transporting* also uses several other high-level plans, making it the most complex plan of the system.

Table 3: Experimental results on the high-level plans

Plan	With state caching		Without state caching	
	#paths	time	#paths	time
Navigating	7	11s	7	11s
Turning	17	12s	24	12s
Searching	55	12s	877	12s
Delivering	373	14s	47185	126s
Accessing	1841	18s	7121	22s
Sealing	1841	18s	7121	22s
Fetching	4105	37s		timeout
Transporting	59161	568s		timeout

Both the low-level and high-level plans are equipped with several failure handlers. These are often organized hierarchically, i. e. when one handler is unable to deal with the underlying problem, it throws a new failure which is handled by a higher-level handler. The deepest handler hierarchies are found in the Searching and Fetching plan with 5 nested failure handlers each.

There are a total of 17 atomic actions, 4 low-level and 8 high-level plans. Each atomic action has between 1 and 3 possible failures with some overlap between actions, for a total number of 13 distinct failure types. The high-level plans have a differing number of arguments, ranging from 1 to 6. There are also 7 reasoning subroutines which decide on certain arguments for some of the plans. All plans combined amount to 2284 lines of bytecode.

4.3.2.2 *Experimental Results*

We substituted all low-level actions and reasoning subroutines according to our approach in Section 4.3.1.2. We then executed our extended version of SEECER on the resulting code. SEECER was configured to report all top-level failures and not terminate after the first find. We evaluated each high-level plan on its own, implicitly also covering all low-level plans and atomic actions. In addition, the arguments for each high-level plan were kept fully symbolical, considerably adding to the complexity.

We found unhandled top-level failures in all eight plans. Some of these would have been easy to find without formal methods as well, e. g. when handlers for certain failures were simply missing. Other failures would be a lot harder to spot manually or via simulation-based testing. For instance, some top-level failures only occurred when an action inside a failure handler itself also failed. There were also cases where several handlers were unable to properly handle a failure until it reached the top level. Our first research question has therefore been answered positively. Our evaluation strongly suggests that symbolic fault injection is a well-suited tool for plan-based robotics. This leaves only the runtime question to be answered.

Table 3 summarizes the runtime results of our evaluation. We report the number of symbolic paths and the total runtime for all eight high-level plans. To show the effect of our proposed state caching technique, we report the results both with and without state caching enabled. The columns of Table 3 report (from left to right) the plan under verification, the number of symbolic paths with state caching enabled, the runtime with state caching enabled and then both metrics when state caching was disabled.

As expected, the runtime for each plan correlates with the number of symbolic paths that are explored. Both metrics depend on the underlying complexity of the plans. Simpler plans, such as Navigating, lead to only few paths. The majority of its $11s$ runtime are not even used for symbolic execution, but rather for the initial setup and compilation of the plan into bytecode. The Transporting plan, on the other hand, uses all other high- and low-level plans. Therefore it is the most complex out of all evaluated plans by far. This becomes apparent in the large number of symbolic paths even with state caching enabled. Nonetheless, its runtime is still below 10 minutes, which is perfectly acceptable for a complete analysis with all possible arguments and a complete list of top-level failures. The analysis for all other plans was finished in under a minute. The two rightmost columns of the table show the effect of disabling our state caching technique. The influence of state caching is clearly visible when looking at the Delivering, Fetching and Transporting plan. Without the optimization, the number of paths of the Delivering plan increased by a factor of over 100 which lead to a runtime increase of 900%. The Fetching and Transporting plans did not finish within the 1 hour time limit, which means a runtime increase of at least 9729% for the Fetching plan. For the other plans, the effects of state caching were less apparent, but nonetheless always positive or at least neutral. We did not observe a case were the slight overhead of state caching actually impacted the runtime negatively.

Overall, our proposed approach was able to completely analyse typical CRAM plans within a short time, for most plans within less than a minute. For the more complex plans this is primarily thanks to our proposed state caching technique.

4.4 CONCLUSION & FUTURE WORK

In this chapter, we presented our approaches for symbolic verification of high-level robotic plans and their integration in our tool SEECER. Our methodology is built on symbolic execution to enable a complete analysis of the underlying CPL plan and all possible execution traces. We presented an interface to connect the CPL plan directly to the environment model. Apart from an environment model in CPL we also investigated the usage of the DEC as a language for environment modelling. Here, we presented a verification methodology for action sequences in pure DEC as well as an integration between DEC reasoning and symbolic execution. Finally, we used symbolic fault injection to explore the failure handling of CPL plans. Our method is able to find unhandled low-level failures under a general worst-case assump-

tion. The experimental evaluations prove the applicability of our approaches. We were able to find errors in several CPL plans.

While SEECER is able to handle realistic CPL plans, its scalability could still be improved. Future work could therefore include some existing modifications for symbolic execution, such as state merging [55] or state subsumption [8]. Especially the integration of those methods with the different environment models are an open research question. Alternatively, scalability could be improved through a combination of concrete and symbolic execution, i. e. concolic execution [91]. This method exchange some completeness of the verification result for a higher scalability. Again, the main challenge here is the integration with the environment model. Furthermore, the environments considered so far focus on a single actor. An interesting direction for future work could be the inclusion of other actors in the environment, such as humans or other robots. While this should already be partially possible through non-deterministic environment models, a specialised approach might still be worthwhile.

5 FORMAL METHODS FOR MODELLING ASSISTANCE

The main challenge of modern autonomous robots is the increasing complexity and uncertainty of their environment. In planning and verification, interactions with the environment play a major role. Therefore, formal models of the environment are necessary to enable reasoning about the robotic plan and its actions. These formal environment models should ideally be simple and abstract. But at the same time they should accurately reflect the behaviour of the real world to achieve meaningful planning and verification results. Building simple, yet accurate models is a challenging and error-prone manual task. In this chapter, we present novel techniques to assist the model designer in this task and therefore enable them to build better and more accurate models.

One of the major challenges when building abstract models is to capture the inherently continuous real world in a discrete logical model. In Section 5.1 we introduce a technique to learn SMT($\mathcal{LRA}$) formulae from example data. These formulae can then be used to find regions in the continuous environment space which exhibit a common behaviour.

In Section 5.2 we introduce a technique to aid in the debugging of existing formal environment models. Our approach compares the formal model to a simulation engine and tries to find discrepancies in the behaviour of the two. The results can then be used to improve the formal model or at least make its limitations explicit.

Finally, Section 5.3 concludes this chapter and presents directions for future work.

5.1 CLUSTERING-GUIDED SMT($\mathcal{LRA}$) LEARNING

Since most logical formalisms for the modelling of robotic environments are discrete, it is usually necessary to partition the environment into a finite set of distinct regions. One way to express regions is the formulation as SMT constraints. An SMT formula can divide a state space into two regions, one in which it is satisfied and one where it is not. SMT constraints also fit nicely into our previously presented approach on symbolic execution, which uses an SMT solver at its core. Writing SMT models by hand is of course possible, but time-consuming and error-prone. Nevertheless, in many cases, both satisfying and unsatisfying examples of model configurations can be extracted from measurements of the modelling domain. In these cases, the actual modelling task can be automated by an approach called *concept learning*. Concept learning has a long history in artificial intelligence, with Probably Approximately Correct (PAC) learning [109], inductive logic programming [77], and constraint programming [15]. These approaches usually focus on pure Boolean descriptions, i. e. SAT formulae. More recently, [51] introduced

SMT($\mathcal{LRA}$) learning, which is the task of learning an SMT($\mathcal{LRA}$) formula from a set of satisfying and unsatisfying examples.

Alternatively, SMT($\mathcal{LRA}$) learning can also be formulated as a variation on the programming by example problem known from the Syntax-Guided Synthesis (SyGuS) framework. Most solvers in this area (e. g. [7, 12, 89, 107]) are based on enumeration of possible solutions to be able to tackle a wide variety of syntactic constraints. This does, however, lead to overly complicated and inconvenient reasoning on continuous search spaces such as SMT($\mathcal{LRA}$). Apart from that, the problems have further subtle differences, e. g. accuracy of the solutions has higher significance in the concept learning setting.

On top of defining the SMT($\mathcal{LRA}$) learning problem, [51] also introduced an exact algorithm called INCAL. As the first of its kind, INCAL naturally comes with certain drawbacks in terms of runtime and is therefore not applicable to learn large models, which are required by most real-world concept learning applications (e. g. [50, 70]).

Our contribution in this work is a novel approach for SMT($\mathcal{LRA}$) learning which uses *Hierarchical Clustering* on the examples to guide the search and thus speed up the model generation process. We call our general approach SHREC (SMT($\mathcal{LRA}$) learner with hierarchical example clustering) and introduce two algorithms SHREC1 and SHREC2 based on this idea. SHREC1 aims at a higher accuracy of the solution and therefore requires more runtime than SHREC2. SHREC2 instead follows a very fast and scalable method with minor losses of accuracy. Therefore, we provide the users, i. e. the model designers, with the possibility to choose between maximizing the accuracy of the learned model or improving runtime of the generation process so that also larger models can be learned in a reasonable time frame.

This section describes the learning framework SHREC. The application to the robotics domain is left for future work.

We first introduce the SMT($\mathcal{LRA}$) learning problem, INCAL and hierarchical clustering in Section 5.1.1. In Section 5.1.2 we introduce our main idea and the algorithm of SHREC1. The scalability improvements and the algorithm of SHREC2 follow in Section 5.1.3. Finally, Section 5.1.4 presents our experimental evaluation.

5.1.1 *Background*

In this section, we give an overview of relevant related work and introduce concepts that we utilize in the remainder of this work.

5.1.1.1 *SMT($\mathcal{LRA}$) Learning*

The problem of SMT($\mathcal{LRA}$) learning has first been introduced in [51]. The goal is to find an SMT($\mathcal{LRA}$) formula which describes some system in the real world. However, no formal representation of the system is available. Instead, a set of measurements is given. In the following, these measurements are called examples. It is further assumed, that there exists an SMT($\mathcal{LRA}$)

formula ϕ^* that accurately describes the system. The problem of SMT(LRA) learning is now defined as follows:

Definition 6. *Given a finite set of Boolean variables $B := \{b_1, \ldots, b_n\}$ and a finite set of real-valued variables $R := \{r_1, \ldots, r_m\}$ together with a finite set of examples E. Each example $e \in E$ $a_e, \phi^* a_e$ is a pair of an assignment and a label. An assignment a_e $B \cup R \mapsto \{\top, \bot\} \cup \mathbb{R}$ maps Boolean variables to $\top$ or $\bot$, and real-valued variables to real-valued numbers. The label $\phi^* a_e$ is the truth value obtained by applying a_e to ϕ^*. We call an example* positive *if $\phi^* a_e$ $\top$ and* negative *otherwise. We denote the sets of positive and negative examples by $E^\top$ and $E^\bot$, respectively.*

The task of SMT(LRA) learning is to find an SMT(LRA) formula ϕ which satisfies all elements in $E^\top$, but does not satisfy any element in $E^\bot$, which can be written as $\forall e \in E$ ϕa_e $\phi^* a_e$.

Example 18. *Consider the SMT(LRA) formula*

$$\phi^* b_1, r_1 \quad \neg b_1 \vee -0.5 \cdot r_1 \leq -1 \wedge b_1 \vee 1 \cdot r_1 \leq 0$$

A possible set of examples would be

$$E \quad \{\{b_1 \mapsto \top, r_1 \mapsto 0\}, \bot, \{b_1 \mapsto \top, r_1 \mapsto 2.5\}, \top,$$
$$\{b_1 \mapsto \bot, r_1 \mapsto 2\}, \bot, \{b_1 \mapsto \bot, r_1 \mapsto -0.6\}, \top\}$$

We call an algorithm that tackles the task of finding an unknown SMT(LRA) formula to a given set of examples, i. e. finding a solution to an instance of the aforementioned problem, *learner*.

Each learner must operate on a given set of possible target formulae, called the *hypothesis space* Φ. Similar to [51], we focus on *CNF* formulae as our hypothesis space.

Definition 7. *We define the* cost *c of a CNF formula with a given number of clauses k and (not necessarily unique) linear constraints h as c $w_k \cdot k$ $w_h \cdot h$, where w_k and w_h are weights associated with clauses and linear constraints, respectively. The cost is a measure for the size and complexity of a formula and can be tuned to focus more on clauses or linear constraints.*

A learner tries to find an SMT(LRA) formula $\phi \in \Phi$. We say that an example e satisfies a formula ϕ iff ϕa_e $\top$ and is consistent with ϕ iff ϕa_e $\phi^* a_e$. Using these definitions, the goal of SMT(LRA) learning is to find a formula ϕ that is consistent with all examples, i. e. as mentioned before, one that is satisfied by all positive examples and unsatisfied by all negative ones.

Example 19. *Consider the example set E from Example 18 again. A possible CNF solution to those examples would be*

$$\phi \quad b_1 \vee 0.5 \cdot r_1 \leq -0.25 \wedge \neg b_1 \vee -1 \cdot r_1 \leq -2.1$$

Obviously, ϕ^* is also a feasible solution, but might not be found by the learner which only knows about the example set E.

Since ϕ^* is not known to the learner and the example set E is usually non-exhaustive, it can not be expected that the learner finds a model equivalent to ϕ^*. It should, however, be as close as possible. This leads to the measure of *accuracy*.

Definition 8. *Given two example sets E_{train} and E_{test} which were independently sampled from the (unknown) SMT($\mathcal{LRA}$) formula ϕ^*, the accuracy of a formula ϕ, which was learned from E_{train}, is the ratio of correctly classified examples in E_{test}.*

Generally, finding any formula for a given example set is not a hard problem. One could construct a simple CNF that explicitly forbids one negative example in each clause and allows all other possible assignments. However, such a formula would have numerous clauses and would not generalize well to new examples, yielding a low accuracy. To avoid such cases of overfitting, a smaller target formula, i. e. one with lower cost, should generally be preferred over a larger i. e. more expensive one.

5.1.1.2 *INCAL*

In addition to introducing the problem of SMT($\mathcal{LRA}$) learning, [51] also presented the first algorithm to tackle it, called *INCAL*. INCAL addresses the SMT($\mathcal{LRA}$) learning problem by fixing the number of clauses k and the number of linear constraints h and then encoding the existence of a feasible CNF with those parameters in SMT($\mathcal{LRA}$). If no such formula exists, different values for k and h need to be used. The order in which to try values for k and h can be guided by the cost function $w_k \cdot k \; w_h \cdot h$.

INCAL's SMT encoding uses Boolean variables to encode which clauses contain which literals, real variables for the coefficients and offset of all linear constraints, and Boolean auxiliary variables encoding which linear constraint and clause are satisfied by which example. It consists of the definition of those auxiliary variables and a constraint enforcing the consistency of examples with the learned formula. To cope with a high number of examples, INCAL uses an iterative approach and starts the encoding with only a small fraction of all variables. After a solution consistent with this subset has been found, additional conflicting examples are added.

The complexity of the learning problem does however not exclusively stem from the size of the input. Another, arguably even more influential factor is the complexity of the learned formula. If an example set requires numerous clauses or linear constraints, it will be much harder for INCAL to solve.

So far, we have discussed the state-of-the-art related work in SMT($\mathcal{LRA}$) learning. In this section, we present a new SMT($\mathcal{LRA}$) learner which incorporates a *hierarchical clustering* technique. To keep this thesis self-contained, we give some preliminaries about clustering in the following section.

5.1.1.3 *Hierarchical Clustering*

In machine learning, the problem of clustering is to group a set of objects into several clusters, such that all objects inside the same cluster are closely

related, while all objects from different clusters are as diverse as possible (cf. [65] for an overview). To describe the similarity between objects, a *distance metric* is needed.

Often, objects are described by means of a vector $v_1, \ldots, v_n$ of real values. Typical distance metrics of two vectors v, w are (1) the *Manhattan distance* (L_1 norm)

$$dist v, w \quad \sum_{i1}^{n} |v_i - w_i|,$$

(2) the *Euclidean distance* (L_2 norm)

$$dist v, w \quad \sqrt{\sum_{i1}^{n} v_i - w_i{}^2},$$

or (3) the L norm

$$dist v, w \quad max|v_i - w_i|.$$

A common approach to clustering is *hierarchical clustering* [96]. The main idea of hierarchical clustering is to build a hierarchical structure of clusters called a *dendrogram*. A dendrogram is a binary tree annotated with distance information. Each node in the dendrogram represents a cluster. Each inner node thereby refers to the union of the clusters of its two children; with leaf nodes representing clusters that contain exactly one vector. This way, the number of contained vectors per node increases in root direction with the root node itself containing all vectors given to the clustering algorithm. Each inner node is also annotated with the distance between its two children. In graphical representations of dendrograms, this is usually visualized by the height of these nodes.

Example 20. *An example dendrogram can be seen in Fig. 21. The dendrogram shows a clustering over six input vectors, labelled A to F. The distance between nodes can be seen on the y-axis. For instance, the distance between vectors* $\{B\}$ *and* $\{C\}$ *is* 1*, while the distance between their combined cluster* $\{B, C\}$ *and vector* $\{A\}$ *is* 2*.*

In this section, we will focus on *agglomerative hierarchical clustering* [96], which builds the dendrogram by assigning each vector to its own cluster and then combines the two closest clusters until a full dendrogram has been built.

To combine the two closest clusters, it is necessary to not only measure the distance between two vectors but also between larger clusters. To this end, a *linkage criterion* is needed. Given two clusters c and d, some established linkage criteria are (1) the *single linkage* criterion, which picks the minimum distance between two vectors from c and d, (2) the *complete linkage* criterion, which picks the maximum distance between two vectors from c and d, or (3) the *average linkage* criterion, which takes the average of all distances between vectors from c and d.

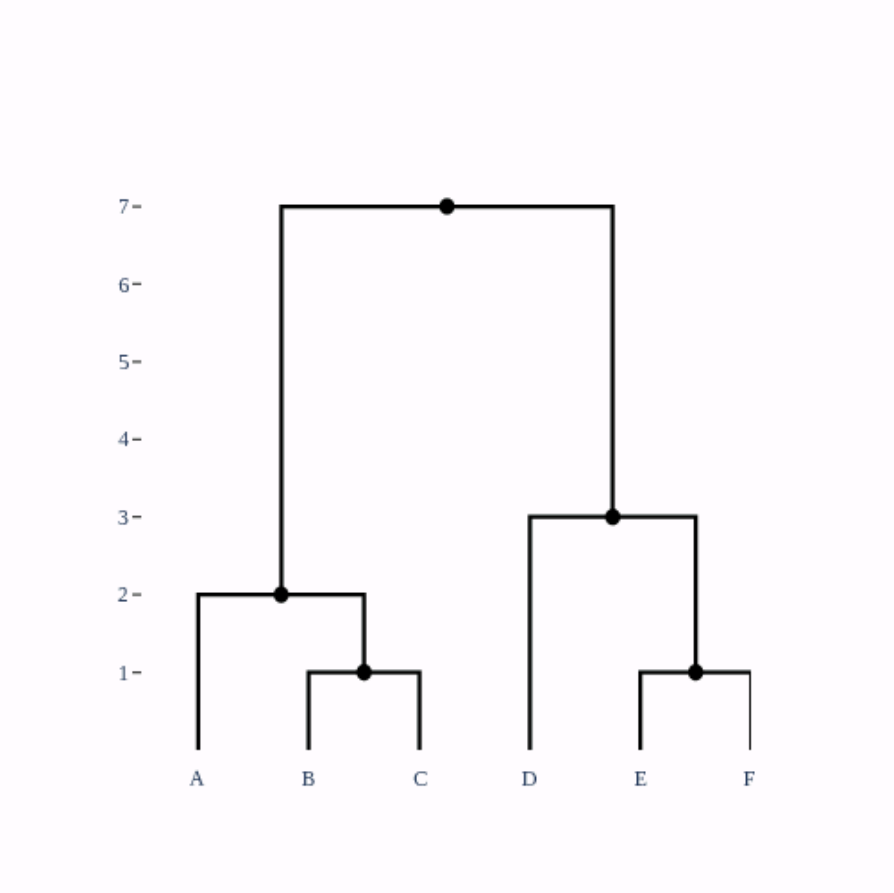


Figure 21: A simple dendrogram

Most combinations of distance measure and linkage criterion can be applied to a given hierarchical clustering problem. The results may, however, vary heavily depending on the application.

To obtain a concrete clustering from a dendrogram, one fixes a *distance threshold*. The final clustering is then made up of the nodes whose distances lie just below the distance threshold and whose parent nodes are already above it. In graphical representations, the distance threshold can be indicated by a horizontal line, making the clusters easily visible. Alternatively, the number of clusters can be fixed and the distance threshold is chosen accordingly.

Example 21. *Fig. 22 shows the dendrogram from the previous example with two distance thresholds. The dashed line represents a distance threshold of* 3.5*. Following this threshold, the dendrogram would be split into the two clusters* $\{A, B, C\}$ *and* $\{D, E, F\}$*. Using a smaller distance threshold of* 2.5*, indicated by the dotted line, would result in the three clusters* $\{A, B, C\}$*,* $\{D\}$*, and* $\{E, F\}$*.*

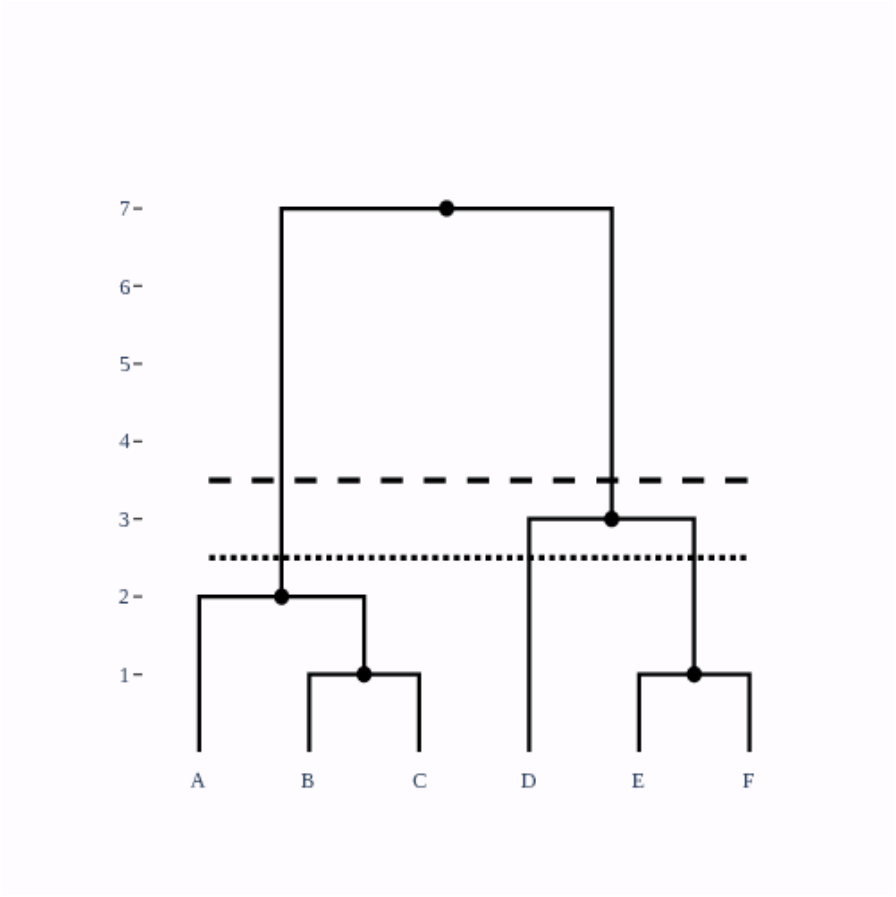


Figure 22: Dendrogram with distance thresholds

5.1.2 *Hierarchical Clustering for SMT(LRA) Learning*

In this section, we introduce our novel SMT(LRA) learner. We describe how the hierarchical clustering is used to guide its search and discuss the resulting algorithm which we call SHREC1. We start with the general idea in Section 5.1.2.1, followed by the algorithm in Section 5.1.2.2 and finally optimizations in Section 5.1.2.3. In Section 5.1.3 we present the algorithm SHREC2 to trade-off some accuracy for a further increase of scalability.

5.1.2.1 *Main Idea*

The main scalability problem of exact approaches for SMT(LRA) learning lies in the large combined encoding that is needed to describe a full CNF. This encoding quickly becomes hard to solve for SMT solvers when the number of clauses and linear constraints is increased. We, therefore, propose to not learn the target CNF as a whole, but rather to learn single clauses and then combine them into the target formula.

When looking at the structure of CNF formulae, it becomes apparent that positive examples need to satisfy all individual clauses, while negative ones only need to unsatisfy a single one. If one had a perfect prediction, which

negative examples belong to which clause, one could simply learn each clause on its own, using a simpler encoding, and still obtain an exact solution. But even an imperfect prediction, which needs some additional clauses, would yield a correct and relatively small solution.

We propose a novel heuristic that produces such a prediction using agglomerative hierarchical clustering. The clustering algorithm partitions the negative examples into groups of closely related examples given their values in the assignment a_e. This is due to the intuition that it is easier to find a single clause for a set of closely related examples as opposed to an arbitrary one. The reason to use hierarchical clustering as opposed to other clustering algorithms is the ability to seamlessly adjust the number of clusters and thus the number of clauses in the target formula.

To obtain a suitable clustering vector, we normalize the examples. For Boolean variables, the values of $\top$ and $\bot$ are replaced with 1 and 0, respectively. The values $a_e r$ of real variables r are translated into the form $\frac{a_e r - r_{min}}{r_{max} - r_{min}}$, where r_{min} and r_{max} are the smallest and highest possible values for variable r, respectively. If those values are not known beforehand, they can simply be estimated from the existing data. This normalization ensures that all feature values lie in the interval 0, 1, which results in each variable having a similar influence on the clustering outcome.

5.1.2.2 *Algorithm SHREC1*

The full algorithm SHREC1 is described in Algorithm 2. The algorithm receives as input a set of examples E and returns a formula ϕ consistent with E. The first step of the algorithm is the function BUILD-DENDROGRAM, which uses agglomerative hierarchical clustering to build a dendrogram from the negative examples. The function uses the normalization procedure described in the previous section. Please note that BUILD-DENDROGRAM is agnostic to specific distance metrics and linkage criteria.

The resulting dendrogram is referred to by its root node N_0. Each subsequent node N_i has a unique, positive index i. As we do not need to distinguish between a node and the set of examples covered by it, we use N_i to refer to both the node N_i and its example set.

The algorithm is composed of several nested loops. The outermost loop (Lines 4-24) searches for a solution with increasing cost. Similar to INCAL, the cost is determined using a linear cost function $w_k \cdot k \;\; w_h \cdot h$. The algorithm starts with the cost value set to w_k in the first iteration, allowing a solution with exactly one clause and no linear constraints. After each iteration, the cost is incremented, increasing the search space.

Since for each cost value multiple combinations of k and h are possible, the next loop (Lines 6-23) starts with $k \quad 1$ and keeps increasing the number of clauses k in each iteration. This, in turn, decreases the number of possible linear constraints. In each iteration, k nodes are selected from the dendrogram through an appropriate distance threshold and stored in the variable *nodes*. The algorithm then tries to find a clause consistent with each node N_i using as few linear constraints as possible. This is done in the function SEARCH-

Algorithm 2 Algorithm SHREC1

Input: Example set E

Output: SMT($\mathcal{LRA}$) formula ϕ

```
1:  function LEARN-MODEL(E)
2:      N_0 ← BUILD-DENDROGRAM(E^⊥)
3:      cost ← w_k
4:      loop
5:          k ← 1
6:          while w_k · k ≤ cost do
7:              φ ← ⊤
8:              nodes ← SELECT-NODES(N_0, k)
9:              h ← 0
10:             valid ← ⊤
11:             for all N_i ∈ nodes do
12:                 cost-bound ← cost − w_k · k − w_h · h
13:                 h′, ψ ← SEARCH-CLAUSE(N_i, cost-bound)
14:                 if ψ ≠ ∅ then
15:                     valid ← ⊥
16:                     break
17:                 else
18:                     φ ← φ ∧ ψ
19:                     h ← h + h′
20:             if valid then
21:                 return φ
22:             else
23:                 k ← k + 1
24:         cost ← NEXT-COST(cost)

25: function SEARCH-CLAUSE(N_i, cost-bound)
26:     h ← 0
27:     while w_h · h ≤ cost-bound do
28:         ω ← ENCODE-CLAUSE(E^⊤ ∪ N_i, h)
29:         ψ ← SOLVE(ω)
30:         if ψ ≠ ∅ then
31:             return h, ψ
32:         h ← h + 1
33:     return h, ∅
```

CLAUSE. If clauses for all nodes could be found within the cost bound, they are combined (Line 18) and the resulting CNF formula is returned. Since each clause satisfies all positive examples and each negative example is unsatisfied by at least one clause, this trivial combination yields a consistent CNF.

The function SEARCH-CLAUSE constitutes the innermost loop of the algorithm. Given a node N_i and the remaining cost left for linear constraints, the function tries to find a clause that is consistent with all positive examples and the

negative examples in N_i. To keep the cost as low as possible, an incremental approach is used again, starting the search with 0 linear constraints and increasing the number of possible linear constraints h with each iteration. To find a clause for a fixed set of examples and a fixed number of linear constraints, an SMT encoding is used in Line 28. This encoding is a simplified version of the encoding from INCAL and uses the following variables: l_b and $\overline{l_b}$ with $b \in B$ encode whether the clause contains b or its negation, respectively; a_{jr} and d_j with $r \in R$ and $1 \leq j \leq h$ describe the coefficients and offset of the linear constraint j, respectively; s_{ej} with $e \in E$ and $1 \leq j \leq h$ is an auxiliary variable encoding whether example e satisfies the linear constraint j.

The overall encoding for a single example e can now be formulated with only two parts, i. e., (1) the definition of s_{ej}, which is identical to INCAL's

$$\bigwedge_{j1}^{h} s_{ej} \iff \sum_{r \in R} a_{jr} \cdot a_e r \leq d_j,$$

and (2) the constraint which enforces consistency of e with the learned clause

$$\bigvee_{j1}^{h} s_{ej} \vee \bigvee_{b \in B} \left((l_b \wedge a_e b) \vee \left(\overline{l_b} \wedge \neg a_e b \right) \right), \quad \text{if } \phi^* a_e$$

$$\bigwedge_{j1}^{h} \neg s_{ej} \wedge \bigwedge_{b \in B} \left((\neg l_b \vee \neg a_e b) \wedge \left(\neg \overline{l_b} \vee a_e b \right) \right), \quad \text{otherwise.}$$

The full encoding is the conjunction of the encodings for all examples in $E^\top \cup N_i$. Like INCAL, SHREC1 also uses an incremental approach. First, we only generate the above encoding for a few examples and then iteratively add more conflicting examples.

The function SOLVE in Line 29 takes an encoding, passes it to an SMT solver, and if a solution to the encoding is found, it is translated back into an SMT($\mathcal{LRA}$) clause. Otherwise, SOLVE returns $\emptyset$.

If a clause could be found within the cost bound (Line 30), it is returned together with the number of linear constraints used. Otherwise, $\emptyset$ is returned together with the highest attempted number of linear constraints.

This basic algorithm can be further improved in terms of runtime and cost by two optimizations described in the next section.

5.1.2.3 *Result Caching and Dendrogram Reordering*

The algorithm SHREC1 as described above suffers from two problems, namely (A) repeated computations and (B) an inflexible search, which we will both discuss and fix in this section.

First, we address issue (A), that SHREC1 re-computes certain results multiple times. When a node is passed to the function SEARCH-CLAUSE together with some *cost-bound*, a consistent clause is searched using up to $\frac{\textit{cost-bound}}{w_h}$ linear constraints. In later iterations of the algorithm's main loop, SEARCH-CLAUSE is called again with the same node and higher *cost-bound*. This leads to the

same SMT encoding being built and solved again. To avoid these repeated computations, each node caches the results of its computations and uses them to avoid unnecessary re-computation in the future.

Second, SHREC1 never modifies the initial dendrogram during the search, making the approach inflexible. We address this issue (B) in the following. If the initial clustering assigns only a single data point to an unfavourable cluster, this might lead to a much larger number of clauses needed to find a consistent formula. This, in turn, leads to a lower accuracy on new examples as well as a higher runtime. To counteract this problem, we apply a novel technique, which we call *dendrogram reordering*: whenever a clause ψ has been found for a given node N_i and some number of linear constraints h, it might be that ψ is also consistent with additional examples, which are not part of N_i, but instead of some other node N_j. To find such nodes N_j, a breadth-first search is conducted on the dendrogram. If some node N_j has been found such that $\forall e \in N_j \;\; \psi a_e \;\; \phi^* a_e$, the dendrogram is reordered to add N_j to the sub-tree under N_i. This does not increase the cost of N_i, because the new examples are already consistent with ψ, but might reduce the cost of N_j's (transitive) parent node(s).

Fig. 23 illustrates the reordering procedure, which consists of the following steps: (1) Generate a new node N_k and insert it between N_i and its parent. Consequently, N_k's first child node is N_i and its parent node is N_i's former parent node. Set N_k's cached clause to ψ. (2) Remove N_j and its whole sub-tree from its original place in the dendrogram and move it under N_k as N_k's second child node. (3) To preserve the binary structure of the dendrogram, N_j's former parent node must now be removed. The former sibling node of N_j takes its place in the dendrogram.

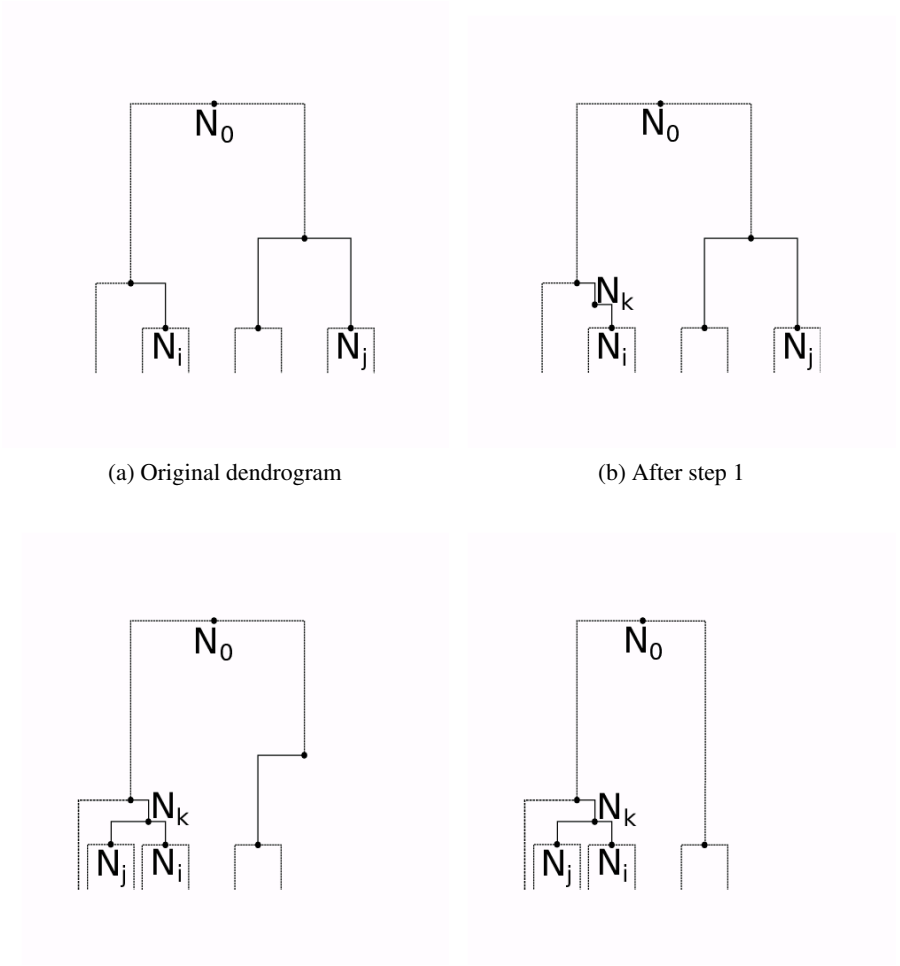
This way, additional examples can be assigned to an already computed clause, inherently reducing the complexity in other parts of the dendrogram. Consequently, the reordering can only decrease the overall cost of the dendrogram and never increase it. Therefore, dendrogram reordering can handle imperfect initial clusterings by dynamically improving them.

5.1.3 *Improving Scalability through Nested Dendrograms*

In the previous section, we introduced a novel SMT($\mathcal{LRA}$) learner with improved runtime compared to INCAL (as we will demonstrate by an experimental evaluation in Section 5.1.4) without a significant impact on the quality, i. e. the accuracy of the resulting formulae. In real-world applications, however, an even faster and more scalable algorithm might be preferred, even with minor losses of accuracy. In this section, we propose a technique for *nested hierarchical clustering* to realize this trade-off. We call this algorithm SHREC2.

5.1.3.1 *Main Idea*

While SHREC1 is already expected to reduce the runtime of the SMT($\mathcal{LRA}$) learner, it still has to solve relatively complex SMT constraints to find a con-



(a) Original dendrogram

(b) After step 1

(c) After step 2

(d) Final result

Figure 23: Dendrogram reordering

sistent clause. To further improve runtime, we again reduce the complexity of these SMT solver calls. The algorithm SHREC2 starts just like SHREC1 by clustering the negative examples and then searching for clauses consistent with the different clusters. However, instead of searching for consistent clauses through an SMT encoding, SHREC2 also clusters the positive examples, ultimately leaving only the learning of single linear constraints to the solver. This is realized through a simpler encoding, shifting the algorithm's overall complexity from exponential to polynomial runtime.

When searching for a single clause, negative examples must not satisfy any literal of the clause, while positive examples only have to satisfy a single literal each. This fact can now be used to learn literals one by one. To this end, *nested dendrograms* are introduced.

We, therefore, extend our definition of dendrograms from the previous sections. A dendrogram that clusters negative examples like the one used in SHREC1 is called a *negative dendrogram* from now on. Its nodes are called *negative nodes* denoted as $N_i^{\perp}$. In SHREC2, we also use *positive dendro-*

grams, which analogously cluster the positive examples. Each node $N_i^{\perp}$ of the negative dendrogram is assigned a new positive dendrogram $N_{i,0}^{\top}$. Each positive node $N_{i,j}^{\top}$ holds a set of positive examples from $E^{\top}$ which again are being clustered just like their negative counterparts.

Given a negative node $N_i^{\perp}$ and some number h of linear constraints, SHREC2 first finds all Boolean literals that are consistent with the examples in $N_i^{\perp}$. Because the cost function is only dependent on the number of clauses and linear constraints, these Boolean literals can be part of the clause without increasing the cost. Then, all positive examples that are inconsistent with any of the Boolean literals are determined. These examples constitute $N_{i,0}^{\top}$. The positive dendrogram under $N_{i,0}^{\top}$ is built in the same manner as the negative dendrogram, using the same normalization scheme. Values of Boolean variables are however left out of the clustering.

To find a set of linear constraints that are consistent with the remaining positive examples as well as the negative examples in $N_i^{\perp}$, an encoding is generated for each of the top h nodes from $N_{i,0}^{\top}$ matching them with individual linear constraints.

5.1.3.2 *Algorithm SHREC2*

Algorithm 3 describes the algorithm SHREC2. The main function LEARN-MODEL is identical to the one in Algorithm 2. The difference here can be found in the function SEARCH-CLAUSE, which tries to learn a clause given a set of negative examples and a cost bound.

The function starts by computing the set L of all Boolean literals that are consistent with all negative examples in $N_i^{\perp}$ (Line 4). It then computes the subset E' of all positive examples not consistent with any literal in L (Line 7). These remaining examples need further literals to be consistent with the clause. Consequently, if E' is already empty at this point, the disjunction of the literals in L is already a consistent clause and can be returned.

Otherwise, additional literals are needed. Because any further Boolean literals would be inconsistent with the negative examples, linear constraints are needed. To find a reasonable assignment of examples in E' to linear constraints, hierarchical clustering is used again. Instead of clustering the negative examples, the algorithm clusters the positive ones in E'. Since Boolean values have no influence on the linear constraints, they are not used in this clustering.

The remainder of the algorithm is now very similar to the process in the main function. The algorithm increases the number of halfspaces in each iteration, starting at 1, until a solution has been found or the cost bound has been reached. In each iteration, the top h nodes from the positive dendrogram are selected. For each node $N_{i,j}^{\top}$, the algorithm tries to find a linear constraint for the examples in $N_i^{\perp}$ and $N_{i,j}^{\top}$ via an encoding. If no such linear constraint exists, the algorithm retries with an increased h. If linear constraints could be found for all nodes, a disjunction of those linear constraints and the literals in L is returned as a consistent clause.

Algorithm 3 Algorithm SHREC2

Input: Example set E
Output: SMT($\mathcal{LRA}$) formula ϕ

```
1: function LEARN-MODEL(E)
2:     ...                                          ▷ identical to SHREC1

3: function SEARCH-CLAUSE(N_i^⊥, cost-bound)
4:     L ← {b ∈ B | ∀e ∈ N_i^⊥  a_e b  ⊥} ∪
           {¬b | b ∈ B, ∀e ∈ N_i^⊥  a_e b  ⊤}
5:     ψ ← ⋁_{l∈L} l
6:     ψ′ ← ψ
7:     E′ ← {e ∈ E^⊤ | ψ a_e  ⊥}
8:     if E′  ∅ then
9:         return 0, ψ
10:    N_{i,0}^⊤ ← BUILD-DENDROGRAM E′
11:    h ← 1
12:    while w_h · h ≤ cost-bound do
13:        ψ ← ψ′
14:        nodes ← SELECT-NODES N_{i,0}^⊤, h
15:        valid ← true
16:        for all N_{i,j}^⊤ ∈ nodes do
17:            ω ← ENCODE-LC N_i^⊥ ∪ N_{i,j}^⊤
18:            θ ← SOLVE ω
19:            if θ  ∅ then
20:                valid ← false
21:                break
22:            else
23:                ψ ← ψ ∨ θ
24:        if valid then
25:            return h, ψ
26:        else
27:            h ← h  1
28:    return h, ∅
```

The encoding for a single example $e \in E$ is a simplified version of the one used in SHREC1, which uses variables a_r and d, describing the coefficients and offset of the linear constraint, respectively. The encoding now only consists of a single constraint per example:

$$\sum_{r \in R} a_r \cdot a_e r \bowtie d$$

where $\bowtie$ is $\leq$ if $\phi^* a_e \quad \top$ and $>$ otherwise. The full encoding is again the conjunction of the encodings for all examples. Like in INCAL and SHREC1, examples are also added iteratively. Please note that the encoding of SHREC2

is only a linear program instead of a more complex SMT($\mathcal{LRA}$) encoding, making it solvable in polynomial time.

SHREC2 also uses result caching and dendrogram reordering in both levels of dendrograms. Additionally, the positive dendrograms are computed only once for each negative node $N_i^{\perp}$ and are immediately cached for faster access.

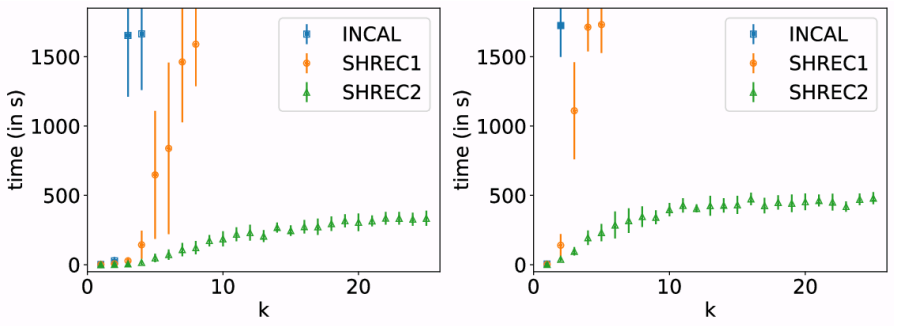
5.1.4 *Experimental Evaluation*

In this section, we evaluate the capabilities and applicability of the proposed algorithms SHREC1 and SHREC2. We have implemented them in Python using the SMT solver Z3 version 4.8.6 and the *scikit-learn* package [84] version 1.3.1 for the hierarchical clustering. To this end, we conducted experiments and compared the results in terms of accuracy and runtime to INCAL. We ran all evaluations on a Linux machine with an Intel CPU with 3.40 GHz (up to 3.80 GHz boost). In the following, we give detailed insight into the experimental setup in Section 5.1.4.1. We present the comparison of our approaches to INCAL in Section 5.1.4.2.

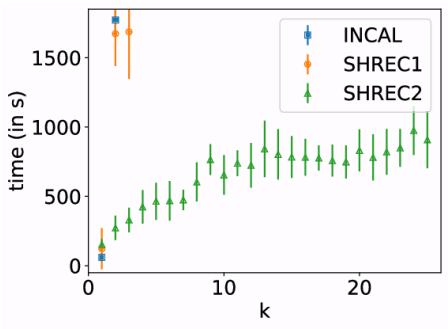
5.1.4.1 *Experimental Setup*

Due to the poor scalability of current approaches, no suitable real-world benchmarks for SMT($\mathcal{LRA}$) learning exist yet. In addition, benchmarks for SMT solving like the SMT-LIB collection are usually either unsatisfiable or only satisfied by few assignments, meaning they do not produce adequately balanced example sets. Therefore, experiments have to be conducted on randomly generated benchmarks. To this end, we use an approach similar to [51]: Given a set of parameters consisting of the number of clauses (k) and linear constraints per clause (h), we generate a CNF formula fitting these parameters. The generation procedure is also given a set of 1000 randomly generated assignments from variables to their respective values. The formula is then generated in such a way, that at least 30% and at most 70% of those assignments satisfy it. To ensure that the formula does not become trivial, it is also required that each clause is satisfied by at least $\frac{30}{k}$% of assignments that did not satisfy any previous clause. This ensures that each clause has a significant influence on the formula and cannot be trivially simplified.

Since the main focus of SHREC is the improved scalability on larger formulae, we (similar to [51]) generated benchmarks with increasing k and h and fixed all other parameters to constant values. All generated formulae have 4 Boolean variables, 4 real variables, and 3 literals per clause. The benchmarks have between 1 and 25 clauses and between 1 and 3 linear constraints per clause, resulting in 75 different parameter configurations. We expect a higher number of clauses or linear constraints per clause to generally result in a harder benchmark. Since we cannot precisely control the difficulty, however, some smaller formulae might turn out to be more difficult than other larger ones. To mitigate these random fluctuations, we generated 10 formulae for each configuration, resulting in a total of 750 benchmarks.



(a) h 1 (b) h 2



(c) h 3

Figure 24: Runtime comparison for different values of h

For each benchmark, 1000 examples were randomly drawn. Boolean variables had an equal probability to be assigned to $\top$ or $\bot$. Real values were uniformly distributed in the interval 0, 1.[1]

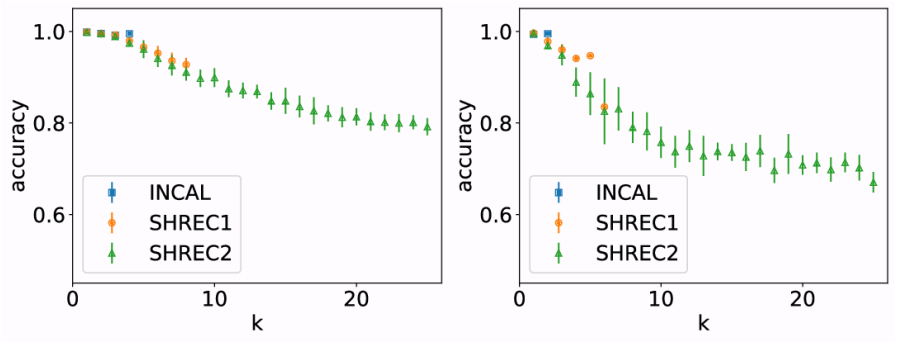
We used INCAL, SHREC1, and SHREC2 to find a CNF formula consistent with all examples. All three algorithms used a cost function with equal weights for clauses and linear constraints (w_k w_h 1). For each run, we measured the runtime and the accuracy on another independent set of 1000 examples. We set a timeout of 30 minutes for each run. This timeout is substantially longer than the one used in [51] and allows us to adequately observe the effect of the different configurations.

In the following section, the results are presented and discussed.

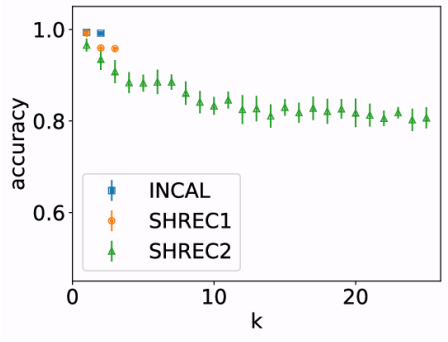
5.1.4.2 *Comparison to INCAL*

As mentioned in Section 5.1.2, SHREC1 and SHREC2 are able to use various distance metrics and linkage criteria in their clustering routine. To determine the most effective combination, we ran some preliminary experiments on a subset of the generated benchmarks. We evaluated the Manhattan distance, Euclidean distance, and the L norm as possible distance metrics and the single, complete and average linkage criteria. Out of the nine possible combinations, the Manhattan distance together with the average linkage cri-

1 Please note that the choice of the interval does not influence the hardness of the learning problem because smaller values do not make the SMT solving process easier.



(a) $h = 1$ (b) $h = 2$



(c) $h = 3$

Figure 25: Accuracy comparison for different values of h

terion performed best. Therefore, this combination is used in the following comparison with INCAL.

Fig. 24 shows the runtime for 1, 2 and 3 linear constraints per clause, respectively. On the x-axis, the number of clauses from $k = 1$ to $k = 25$ is shown. The y-axis shows the runtime in seconds. Each data point covers the runs on the 10 different benchmarks for the respective configuration. The squares, circles, and triangles mark the mean of all 10 runtimes, while the vertical error bars show the standard deviation. Runs that timed out were included in the calculation of mean and standard deviation as if they needed exactly 1800 seconds. If all runs of one configuration timed out, no data point is shown.

As expected, the number of clauses and linear constraints increases the runtime of all three algorithms. However, we can observe that the increase in runtime becomes smaller at a higher number of clauses. INCAL already times out at $k \geq 5$ for benchmarks with a single linear constraint per clause and even at $k \geq 3$ for benchmarks with 2 or 3 linear constraints per clause. SHREC1 is able to handle larger benchmarks better, but still times out at $k \geq 8$, $k \geq 6$ and $k \geq 4$ for $h = 1$, $h = 2$, and $h = 3$, respectively. On instances where neither INCAL nor SHREC1 time out, SHREC1 is consistently considerably faster. SHREC2 is a lot more robust for increasing k and h and does not time out on any of the instances. SHREC2's runtime stays far below that of INCAL and SHREC1 for almost all of the benchmarks. This indicates SHREC2's

superior scalability in terms of runtime, outperforming INCAL and SHREC1 by a large margin.

Naturally, we expect this success to come with a trade-off in the form of lower accuracy. Fig. 25 shows the accuracy for 1, 2 and 3 linear constraints per clause, respectively. As before, each data point shows the mean and standard deviation of 10 benchmarks. Timeouts were not considered in the calculation this time. Configurations with 10 timeouts again have no data point displayed. As expected, the accuracy of all three algorithms is lower for larger problems. This is because a more complicated CNF needs to be found with the same number of examples. One can also observe that SHREC1 and especially SHREC2 suffer more from this decrease in accuracy than INCAL. However, as Fig. 25a shows, SHREC1's accuracy still stays above 95% even for benchmarks with up to 7 clauses.

The decrease of accuracy is only crucial for larger values of k and h, which were not solved by INCAL at all. If given enough time, we can also expect INCAL to show a lower accuracy for these harder benchmarks. For the benchmarks which were solved by INCAL, SHREC1 and SHREC2 stay very close to 100% accuracy, as well. If one wants to compensate for the lower accuracy in other ways, the improved scalability of SHREC1 and SHREC2 could also be utilized to simply incorporate more training examples that can be handled due to the better scalability.

Overall, the experimental results clearly show that SHREC is superior to the state-of-the-art exact approach INCAL in terms of scalability. SHREC1 needs considerably less runtime to learn formulae with only a slight loss of accuracy, while SHREC2 was several magnitudes faster and still kept the accuracy at a reasonable level.

5.2 SIMULATION-BASED DEBUGGING OF FORMAL ENVIRONMENT MODELS

Generalized robotic plans are often built making use of formal environment models. In Section 4.2 we also use formal models for the verification of robotic plans.

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 modelled in terms of formal logic. Another reason is the discretisation 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 behaviour 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. Un-

fortunately, 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 section, we aim to make these discrepancies explicit by combining formal verification techniques with robotic simulation. The main idea is to use our formal verification engine SEECER presented in Chapter 4 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.

In Section 5.2.1 we present our approach to simulation-based debugging and follow with an experimental evaluation in Section 5.2.2.

5.2.1 *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 5.2.1.1. Afterwards, we describe the sampling and confidence calculation in greater detail in Sections 5.2.1.2 and 5.2.1.3.

5.2.1.1 *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.

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 discretisation is usually implemented by modelling 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 $m \quad S_c \mapsto S_d$, 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_{i1}. 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.

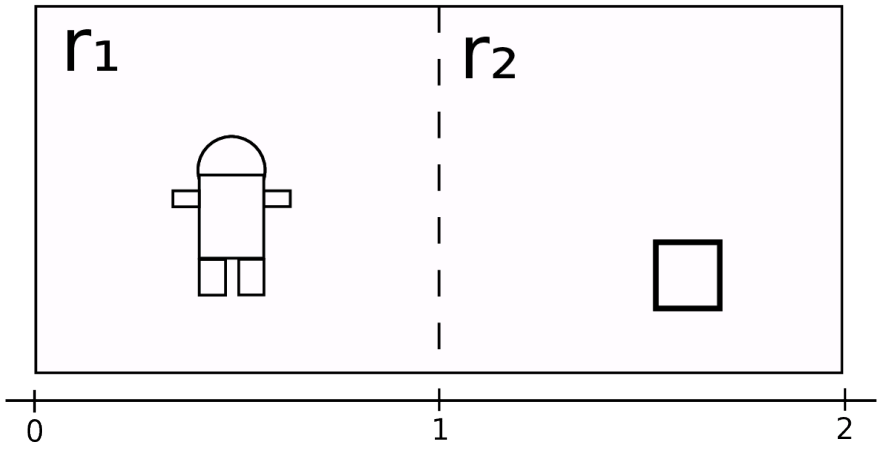


Figure 26: Simple robotic environment

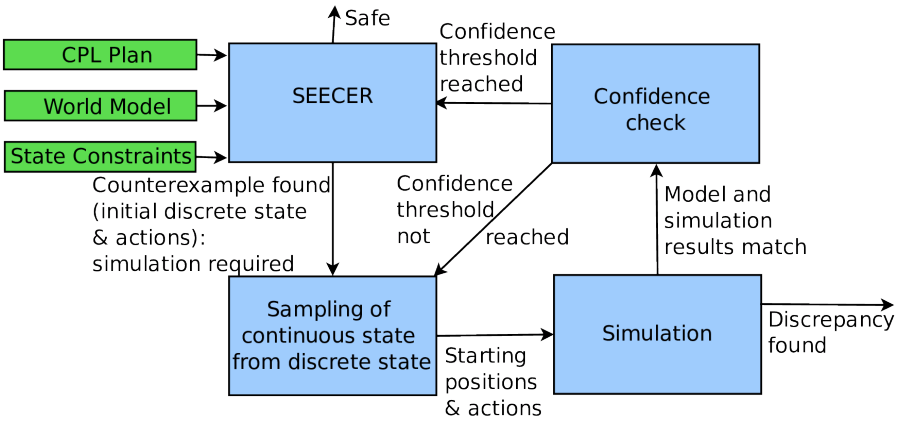


Figure 27: Overview of our debugging approach

Example 22. *Consider the simple environment depicted in Fig. 26. 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.*

Fig. 27 shows the high-level flow of our approach. The input to the procedure consists of three components (green boxes in Fig. 27): 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 Fig. 27), 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 discrete 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 $ms_{c,0} \quad s_{d,0}$ 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 predefined 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.

5.2.1.2 *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 23. *Consider again the environment and plan from Example 22. Sampling an initial concrete state might result in the robot at* $x \quad 0.8$ *and the box at* $x \quad 1.4$. *After setting this state in the simulation and executing the actions, both the robot and box would be at* $x \quad 0.8$. *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 section.

5.2.1.3 *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 $y \quad \sum_{i1}^{n} x_i \beta_i \ \epsilon$, 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 $y \quad X\beta \ \epsilon$, 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 $y \quad Xb \ \epsilon$, 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 24. *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. Let's 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 $\sum_{i1}^{n} \epsilon_i^2 \quad \epsilon'\epsilon$. The respective values of b can be estimated by $b \quad X'X^{-1}X'y$ and the least squared error by $\sigma \quad min\epsilon'\epsilon \quad y - Xb'y - Xb$. 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 $\sum_{i1}^{n} x_{i,min} \cdot b_i - \frac{Z\sqrt[n]{C}\sigma}{\sqrt{s}}, \sum_{i1}^{n} x_{i,max} \cdot b_i \ \frac{Z\sqrt[n]{C}\sigma}{\sqrt{s}}$, 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 25. *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 σ 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.*

5.2.2 *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, 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 CPU with 2.50 GHz clock rate.

In the following Section 5.2.2.1 we present the three robotic plans used in the evaluation. Section 5.2.2.2 contains the experimental results and the discrepancies that have been found. We close with a discussion in Section 5.2.2.3.

5.2.2.1 *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.

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 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.

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.

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.

5.2.2.2 *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

Table 4: 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

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 one 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 discretisation used in the formal model was simply too coarse to accurately reflect the true conditions for visibility and reachability of the spoon. A finer discretisation 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 4. 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.

5.2.2.3 *Discussion*

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.

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 discretisation. 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, such a refinement may sometimes 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.

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.

5.3 CONCLUSION & FUTURE WORK

In this chapter, we presented techniques to assist a model designer in the complex task of devising a logic-based model of a robotic environment. Our first approach takes a set of satisfying and unsatisfying examples to learn an SMT($\mathcal{LRA}$) formula. Compared to the state of the art, we are able to achieve a considerably better scalability with only minor losses in accuracy, as evidenced by our experimental evaluation. We also presented two different algorithms, so the model designer is able to choose between higher scalability or higher accuracy.

The second approach uses our symbolic execution engine SEECER to find execution traces in which a formal model's behaviour differs from that of a simulation engine. 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.

We envision the two approaches presented here as part of a comprehensive design flow for formal models. There are still several pieces missing to this flow, however. The application of our SMT($\mathcal{LRA}$) learner to actual robotic data is still open work. Likewise, other classes of SMT formulae should be considered as well. The selection of useful input examples is an open research question, as well.

Right now, our debugging 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.

For a complete design flow, several other building blocks are still missing, as well. These are the generation of formal descriptions from the identified

regions, an adequate selection of the right abstraction level and an integration in planning and verification tools.

6

COVERAGE-GUIDED FUZZING OF ROBOTIC PLANS

In Chapter 4 we presented different formal methods for the verification of high-level robotic plans. Formal verification is a great way to cover the robotic plan completely including hidden edge and corner cases. However, this completeness comes with the downside of a high runtime and no guaranteed termination. Depending on the complexity of the plan, formal verification methods may not terminate at all or only after an unreasonably long time.

In many other domains, coverage-guided fuzzing has proven to be an effective compromise between hand-written tests and formal verification. In coverage-guided fuzzing, inputs to a program are generated semi-randomly and the correctness of the output is checked automatically. This way, a large number of test cases can be run without manual interaction. During execution, the coverage on the code is measured and used to guide the generation of subsequent inputs. The goal is to maximize the coverage of the generated test cases.

This way, coverage-guided fuzzing is able to test relevant edge cases that a human test engineer may have missed. At the same time, coverage-guided fuzzing can be terminated at any time and has no significant runtime overhead over manual tests.

In this chapter, we present our approach to coverage-guided fuzzing for plan-based robotics. Our contributions are threefold: First, we introduce coverage-guided fuzzing to the domain of plan-based robotics. Secondly, we present a prototypical implementation for the robotic plan language CPL. Finally, we introduce a novel coverage metric for the domain of plan-based robotics that may be used in combination with coverage-guided fuzzing or independently of it.

Our approach builds upon SEECER, except here, we only use SEECER for the discrete execution of CPL plans. No symbolic values and no SMT solver is used. A fuzzer is instrumented to provide SEECER with different initial states of the simulation as input to the plan. During execution the resulting code coverage is measured and fed back to the fuzzer.

Our novel coverage metric measures the percentage of possible actions that have been executed by the plan and thus follows the effect of the plan on its environment more closely than general structural coverage metrics.

In Section 6.1 we introduce our approach to coverage-guided fuzzing for CPL. Section 6.2 presents our novel *action coverage*. Section 6.3 discusses the experimental evaluation of our approach and Section 6.4 concludes this chapter.

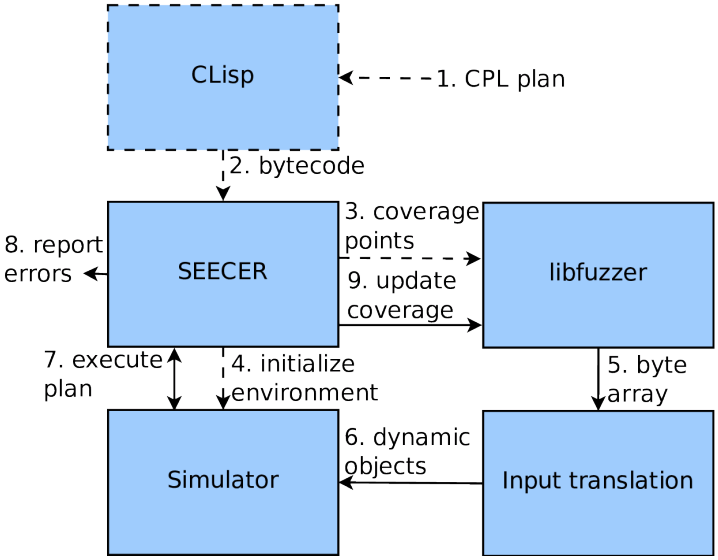


Figure 28: Overview of our fuzzing approach

6.1 COVERAGE-GUIDED FUZZING FOR CPL PLANS

In this section, we introduce our approach to coverage-guided fuzzing of CPL plans. We start with an overview of our methodology in Section 6.1.1. Afterwards, we explain two aspects of our approach in more detail. These are the translation of the fuzzer output to an initial environment state in Section 6.1.2 and the coverage measurement in Section 6.1.3.

6.1.1 *Overview*

In most applications, the fuzzer will provide inputs to a program or function. In the context of plan-based robotics however, the plan will receive inputs from its environment. We therefore propose to use the fuzzer output to generate an environment for the robot. We divide a robots environment into a static and a dynamic part. The static part of the environment is the same for all executions and may e. g. contain walls or larger pieces of furniture. The dynamic part should be different between executions and contains smaller items that are supposed to be manipulated by the robot.

We use an adapted version of SEECER in combination with CLisp and the fast projection simulator for the plan execution and libfuzzer for the input generation. The flow of our program is shown in Fig. 28. It is divided into an initialization phase indicated by dashed arrows and a main loop indicated by continuous arrows. The steps are numbered according to their order.

During initialization, the CPL plan (1) is first parsed and compiled into CLisp bytecode (2). This bytecode is then analysed to find all coverage points. A memory segment is reserved for the respective counters and given to the fuzzer (3). Finally, the simulation is initialized and the static part of the environment is loaded (4).

After the initialization steps are complete, the procedure enters a main loop that repeats the following steps. At first, the fuzzer provides a byte array as input to the plan (5). This byte array is then translated into a set of objects, which are added to the simulation (6). Afterwards, the robotic plan is executed in the simulation environment (7). During execution, the counters of the chosen coverage metric are updated after every instruction. After the execution has finished, the final state of the simulation is checked for erroneous behaviour such as objects in the wrong location. Any errors found are reported to the user (8). Additionally, the coverage is updated in the fuzzer (9) and also reported to the user. Finally, the simulation environment is reset to prepare for the next iteration.

The main loop can run as long as desired by the user. Possible stopping criteria include the number of found errors, a time limit or a coverage limit.

6.1.2 *Initial Environment Setup*

Unlike most applications, plan-based robotics require the fuzzer to provide an initial environment setup instead of an input to a function. In this section, we will cover the translation from generated bytes to this environment setup in more detail. At first, the environment needs to be separated into a static and a dynamic part. Only the dynamic part will change between iterations. The static part remains constant throughout the whole procedure and is therefore independent of the fuzzer output.

For the dynamic part, objects need to be generated with several properties such as their type, position and orientation. Since not all positions within the environment may be eligible to create an object at, we further propose to define regions and reserve part of the generated bytes to first decide the region and then the coordinates within that region.

Depending on the number of regions and types as well as the desired granularity on positions and orientations more than one byte may be necessary to represent an object. With t possible types, r possible regions, p possible positions per region and o possible orientations, the number of bytes b should be chosen such that $256^{b-1} < trpo \leq 256^b$, i. e. the smallest number that will be able to represent all combinations of type, region, position and orientation.

If the fuzzer produces a total number of bytes that is not divisible by b, the remaining incomplete object is discarded.

Example 26. *Consider a simple environment with three tables, which are* 90*cm by* 90*cm. In the initial state, a number of bottles and cups are placed on any of the tables. The test designer chooses a grid with a width of* 20*cm, which results in* $4 \cdot 4$ 16 *possible positions per table. The objects will always stand upright, but may by turned by multiples of* 90 *degrees, resulting in* 4 *possible orientations. With* 2 *types,* 3 *regions,* 16 *positions and* 4 *orientations, there are a total of* 384 *possible configurations per object and two bytes will be necessary to represent an object. When the fuzzer produces* 5 *bytes, only two objects will be instantiated and the last byte is discarded.*

Of course, other properties like dimensions, colour, fill level of containers, etc. may be represented in the same way, when applicable.

6.1.3 *Coverage Measurement*

Our approach needs to measure the code coverage to guide the fuzzer and report it to the user. In this section, we will describe the instrumentalization of SEECER and the coverage measurement in detail.

Since SEECER operates on CLisp bytecode, we will also define our coverage metrics on that bytecode instead of the higher-level CPL plan. We will mainly describe the instruction and branch coverage, but other structural coverage metrics can be added in a similar manner.

Since libfuzzer requires a counter for each coverage point, we will also use this representation internally. During the initialization phase of our approach, the bytecode will be analysed to find the total number of coverage points. For the instruction coverage this simply corresponds to the number of executable instructions. For the branch coverage, the conditional jumps are counted and multiplied by two, since there are exactly two outcomes for each conditional jump. An array of these counters is created and initialized with zeros.

During execution the counter array is updated using an observer pattern. Coverage metrics will register at the interpreter and in turn the interpreter will notify them after each instruction execution. The instruction coverage metric reacts to all instruction executions and increments the respective counter. The branch coverage metric only reacts to conditional jump instructions and increments one of the two respective counters depending on whether the branching condition is true or false.

To measure the total coverage, the number of non-zero entries in the array is divided by the total number of entries.

Example 27. *Consider the bytecode in Fig. 29, which was previously used in Chapter 2. The bytecode is divided into a data section (the unnumbered lines at the top) and a code section (the numbered lines). The code accesses the data through the* `CONST` *instructions in Lines 1, 8 and 12.*

The program requires one integer to be present on the stack. It will then load the first constant, the numeric value 2 *and apply the built-in function* 210*, which is the modulo operation (Line 3). The result is compared to zero (Line 5) and depending on the outcome the execution will jump to Line 11 or proceed with Line 8. Ultimately, the program will return either* `"EVEN"` *or* `"ODD"`*, depending on the value of the input.*

For this program, SEECER will initialize a counter array with 15 *entries for the instruction coverage, since there are* 15 *instructions. The counter array for the branch coverage will have only* 2 *entries, one for each possible result of the* `JMPIF` *instruction in Line 7. The* `JMP` *instruction in Line 10 does not require any coverage points, since it is unconditional.*

Assume that the program is called with an even input. This will execute Lines 1 to 7 and Lines 11 to 15. This results in a total of 12 *executed instructions and a instruction coverage of* $\frac{12}{15}$ 80%*. Of the coverage points for the*

```
(CONST 0) = 2
(CONST 1) = "ODD"
(CONST 2) = "EVEN"
(CONST 3) = IS-EVEN

1  (CONST 0)          ; 2
2  (PUSH)
3  (CALLS2 210)       ; MOD
4  (PUSH)
5  (CALLS1 172)       ; ZEROP
6  (SETVALUE 3)       ; IS-EVEN
7  (JMPIF L11)
8  (CONST 1)          ; "ODD"
9  (PUSH)
10 (JMP L14)
11 L11
12 (CONST 2)          ; "EVEN"
13 (PUSH)
14 L14
15 (SKIP&RET 1)
```

Figure 29: CLisp bytecode example

branch coverage, only the one corresponding to the value $\top$ *is incremented, resulting in a branch coverage of* 50%.

6.2 A COVERAGE METRIC FOR PLAN-BASED ROBOTICS

While general structural coverage metrics like instruction or branch coverage have proven their usefulness, domain-specific functional metrics are often able to follow the intended behaviour of the program more closely. Therefore, in this chapter, we introduce *action coverage* as a natural functional coverage metric for plan-based robotics. The metric is independent of the concrete plan language, but will be presented and evaluated in the context of CPL in this section.

The general idea is to measure which percentage of the possible actions have been executed by the plan. Here, not only the type of the action, but all parameters are considered. This makes the metric neither strictly stronger or strictly weaker than the presented structural coverage metrics. For instance, the same line of code may execute an action with different parameters depending on the value of some variable. The second execution of that line would then increase the action coverage, but not the instruction or branch coverage.

If all parameters of the executable actions are discrete and have sufficiently few values, each possible action parametrization can correspond to a coverage point. The coverage calculation and implementation are straight-forward in this case.

Example 28. *Consider again the simple environment from Example 26 with three tables and two object types. Also consider a two-armed robot acting in this environment. The robot may pick an object from any of the tables or place an object on a table. The action abstracts from the exact position on the table. It is parameterised by its type (*pick *or* place*), the table, the object type and the arm that is used. This allows for a total of* $2 \cdot 3 \cdot 2 \cdot 2$ 24 *distinct actions to be performed, resulting in* 24 *coverage points.*

However, in many cases there will be continuous parameters or ones with a lot of possible values. In these cases a straight-forward approach will still work to some extend, but due to the extremely high or even infinite amount of possible actions, the overall coverage will be either very close to zero or undefined. To avoid this problem, we suggest to form *buckets* of similar actions and create one coverage point per bucket.

A bucket is a set of actions that are sufficiently similar in their parameters. The space of all possible actions should be divided into a finite set of buckets such that each action belongs to exactly one bucket. After an action is executed, the respective bucket is marked as executed. In our implementation of coverage-guided fuzzing, each bucket would have its own counter that is incremented whenever an action from that bucket is executed.

The choice of buckets is highly domain-specific and may depend on the plan and environment under observation. This obviously makes it harder to compare the quality of different plans acting in different environments. Still, the comparability of different test sets for the same plan is preserved and the metric is well suited to guide a fuzzer.

Example 29. *Consider again the environment and actions from the previous example. Now, assume an additional navigation action that will navigate the robot to a continuous coordinate within the room. This results in an infinite number of distinct actions. To reduce the number of coverage points to a manageable amount, the navigation action is divided into* 4 *buckets depending on its target position. There is one buckets for each table and its surrounding area and one bucket for all positions not adjacent to a table. This increases the total number of coverage points to* 28.

Action coverage can be used in combination with coverage-guided fuzzing as presented in the previous section, but also independently. Like other coverage metrics it may be used to judge the quality of hand-written or (semi-)automatically generated test cases.

We believe that action coverage measures the diversity of plan executions more closely than structural coverage metrics, since the focus is on the actual behaviour of the robot in its environment rather than the control flow of the underlying program.

6.3 EXPERIMENTAL EVALUATION

This section describes our experimental evaluation. We evaluate both our approach to coverage-guided fuzzing for plan-based robotics in general and

the combination with action coverage in particular. All experiments were conducted on a Linux machine running an Intel CPU with 2.50 GHz clock rate. In Section 6.3.1 we present the plan and environment that were used for the evaluation. Afterwards, we discuss our results in Section 6.3.2.

6.3.1 *Robotic Plan and Environment*

We evaluate our approach on a CPL plan that is set in a warehouse-inspired environment. The static part consist of a table and a shelf with three boards in a rectangular room. The dynamic part contains a variable number of objects with three types (milk, cereal and bowl). Initially, the objects may be on any of the shelf boards or on the table. The plan is supposed to sort the objects onto the shelf boards. Each object type has a corresponding board on the shelf. It does so by first moving all objects to the table, clearing the shelf in the process, and then moving them to their respective shelf boards. To save trips between the shelf and table, the robot will always transport two objects at once if possible. Due to the width of the shelf, the robot is not able to reach all positions on it from the same point. A series of case distinctions is responsible for picking the right position for the robot to pick or place both of its objects.

In total, the plan involves 1785 bytecode instructions, 52 branching instructions and 6 different action types. These are the *move-torso*, *park-arms*, *detect-objects*, *navigate*, *pickup* and *place* action.

For the action coverage, we decided on a total of 87 buckets. One bucket belongs to each of the move-torso, park-arms and detect-objects actions. The navigate action has 6 buckets, which are distinguished by their target position. The pickup action also has 6 buckets, depending on the arm and the type of the object. Finally, the place action is divided into the remaining 72 buckets, which are distinguished by the arm, the type of the object and the target position.

The initial state of the environment is built using two bytes per object. The first byte decides the type of the object and one of four regions: the top of the table and the top of each of the shelf boards. The second byte is split in half, with the first four bits corresponding to the relative x position and the last four bits to the relative y position of the object within the region. The z position and the orientation are fixed for each region.

6.3.2 *Experimental Results*

In this section we present the results of our experimental evaluation. During execution, we measured the instruction, branch and action coverage. The fuzzer is however only able to consider one coverage metric at once. Therefore, we executed three versions, with each metric being the guiding metric to the fuzzer in one version. To achieve a higher consistency of the results, we executed ten runs per version, for a total of 30 runs. Each run had a time limit of 5 hours.

Table 5: Minimum, maximum and average time to find each error

Error	min	max	avg
Primary table	11s	89s	45s
One too high	10s	122s	57s
Two too high	13s	315s	102s
Secondary table	8s	528s	109s
Shelf edge	8s	511s	169s
One too low	125s	3348s	714s
Two too low	411s	9517s	2336s

We evaluated the following research questions:

- Is coverage-guided fuzzing able to find relevant errors in robotic plans in a reasonable time?
- How well do the investigated coverage metrics reflect a thorough testing of the robotic plan?
- Which effect does the guiding coverage metric have on the fuzzing process?
- How consistent are the results between runs?

The runs unveiled a total of 7 errors in the plan, which we categorized by their effect on the final environment state.

The *shelf edge error* occurred when an object in the initial state was very close to the back edge of the shelf. This caused it to be occluded by the shelf board. The robot could therefore not detect the object and would not move it. This of course caused an invalid final state, if the object was not initially on its correct shelf board. Additional positions for the detection of objects would be necessary to mitigate this error.

In some cases, objects were left on the table, because they were occluded by other objects and thus not detected in the second part of the plan. We call these errors *primary table error* if the object was on the table in the initial environment state and *secondary table error* if it was moved there. To avoid this error, the detection and moving objects from the table should be repeated until the table is empty.

The final four error categories describe objects that were sorted onto the wrong shelf board. These errors stem from either an internal logic error in the plan or from an inaccurate placing action. Depending on the difference between the expected and actual shelf board, we call these errors *one too high error*, *two too high error*, *one too low error* or *two too low error*.

All seven errors were found in all 30 runs, but the time it took to find each error differed. The minimum, maximum and average times it took to find each error are shown in Table 5. The first column contains the error name,

followed by the minimum, maximum and average time in seconds that it took to find the respective error. The earliest found errors were the shelf edge error and the secondary table error, which were each found after 8 seconds in two different runs. The error that took the most time to be found was the two too low error after 9517 seconds (just over 2h and 38min). This strong difference between error types is also visible in the average times. The two too low error took over 50 times as much time to be found on average than the primary table error. But also the time for each error type differed greatly. This is best seen with the secondary table error, where the maximum time is 66 times as high as the minimum time. The guiding coverage metric had no clear effect on the time it took to find errors.

The coverage metrics increased in different ways during runs, but converged to the same values after 5 hours for all 30 runs. These values were 97.1% branch coverage, 95.0% instruction coverage and 59.3% action coverage. Upon further inspection of the CPL plan, these values were found to be the theoretical maximum due to a small section of unreachable code and several action buckets that could not be executed by the plan. This also showcases that finding suitable buckets is not a trivial problem, since many parameters of the actions are only decided at runtime. And while it was no particular priority for this evaluation, it shows that finding a diverse set of buckets that still allows 100% action coverage is not an easy task.

The amount of time it took to reach those maximum values differed greatly between runs. The branch coverage and instruction coverage always reached their maximum at the same time, even though the increases during the runs were not necessarily synchronous. The fastest time for those two metrics to reach the maximum was 20 seconds and the slowest time 283 seconds. The average time was 98 seconds. The highest action coverage was reached much slower, with a minimum of 2353 seconds, a maximum of 13079 seconds and an average of 6802 seconds. Again, there was no clear effect of the guiding coverage metric.

The vastly slower convergence of the action coverage suggests that it is harder to fulfil than the other two metrics. This also suggests that judging a set of test cases by their action coverage holds them to a higher standard than the branch or instruction coverage. To undermine this statement, we also looked at the number of errors that were found only after the branch, instruction or action coverage had reached their maximum. The reasoning here is that a maximum value of some coverage metric should usually indicate that the test cases cover a high amount of all possible outcomes and additional errors after that are unlikely. So if a lot of errors were found after a coverage metric's maximum was reached, the metric is likely not thorough enough.

Of the 30 total runs, several errors occurred only after the branch and instruction coverage had reached their maximum. These were 5 occurrences of the primary table error, 6 occurrences each of the secondary table error and the two too high error, 10 occurrences of the one too high error, 12 occurrences of the shelf edge error, 25 occurrences of the one too low error and all 30 occurrences of the two too low error. Only 2 occurrences of the

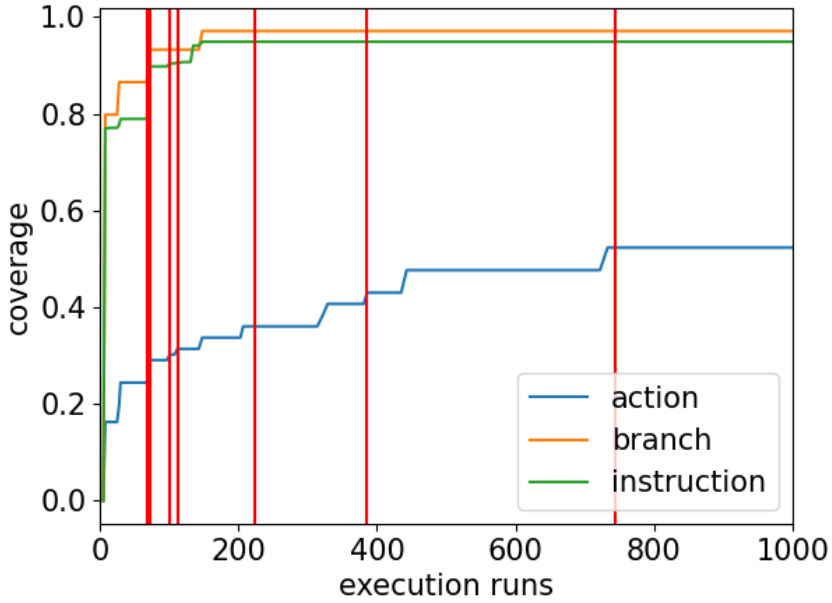


Figure 30: Exemplary coverage development over time

two too low error occurred after the maximum of the action coverage was reached. This clearly shows that the branch and instruction coverage are insufficient for a thorough testing of the robotic plan, while the action coverage had much better outcomes.

Example 30. *To visualize the difference between the metrics, consider Fig. 30 that shows the results of the first run (guided by the instruction coverage). The y-axis shows the coverage for each metric and the x-axis shows the time in seconds. To achieve a better visibility of the results, only the first 1000 seconds of the run are shown. The blue, orange and green line show the development of the action, branch and instruction coverage, respectively. The red vertical lines show points at which an error of each category was found for the first time. The figure shows that the first four errors were found quickly and before the branch and instruction coverage had reached their maximum. The later three errors however were only found afterwards. All seven errors were found before the action coverage reached its maximum, which happened outside of the scope of the graphic.*

With respect to our research questions we can say that coverage-guided fuzzing was able to find relevant errors in the tested robotic plan. In each run 7 errors were found. This is consistent in terms of the final result, but not necessarily in terms of the time needed. The time necessary to find certain errors varied greatly between runs, as can be expected from a semi-random algorithm. We found that the action coverage is a good indicator of the completeness of a test suite, since in most cases, all errors were found when it reached its maximum. The instruction and branch coverage on the other hand did not work well as an indicator, as almost half of all errors were found after

both metrics reached their maximum. This quality of the action coverage metric did however not carry over to its use as a guiding coverage metric. There were no clear differences in the behaviour when a different metric was chosen. Since the action coverage performed well otherwise, this might suggest that the chosen fuzzer is simply not very sensitive to the guiding coverage metric. Overall, both the fuzzing approach and the action coverage have been successful in our evaluation.

6.4 CONCLUSION & FUTURE WORK

In this chapter, we introduced coverage guided fuzzing to the domain of plan-based robotics. We presented our implementation for CPL.

Our approach starts with an initialization phase, which handles the initialization of the fuzzer and the simulation as well as the compilation and analysis of the CPL plan. In the subsequent main loop, the byte array provided by the fuzzer is translated into an initial environment setup and the plan is executed in that environment. During execution, the coverage is measured and fed back to the fuzzer.

In addition to the fuzzing approach, we presented a novel coverage metric for the domain of coverage-guided fuzzing, which measures the percentage of possible actions that have been performed by the plan.

Our experimental evaluation shows that coverage-guided fuzzing is able to find relevant bugs in high-level robotic plans. The novel coverage metric proved useful in judging the quality of a test suite.

For future work, additional coverage metrics should be incorporated. A custom fuzzer backend, e. g. domain specific mutation patterns would also be an interesting research direction.

7 CONCLUSION

Current advances in technology and artificial intelligence pave the way for a new generation of autonomous robots that are far more integrated into human environments than their predecessors. Possible application areas include search and rescue scenarios, household assistance, elderly care or medical procedures.

One promising approach to handle the high complexity of the robots' tasks and environments is plan-based robotics. Here, a high-level plan is responsible for the orchestration and supervision of lower-level modules such as a motion planner, knowledge base or computer vision module. The approaches in this thesis have been implemented for the high-level, Turing-complete plan language CPL.

When robots act in human environments such as households, the safety and correctness of the robotic plan is highly important. Therefore, in this thesis, we investigated several techniques to uncover hidden bugs, prove the correctness of the robotic plan and assist in the planning and verification process.

We presented a framework for the symbolic verification of robotic plans under different environment models. In particular, we enabled an integration between our symbolic execution framework SEECER and the Discrete Event Calculus. We also introduced a technique to find unhandled low-level failures through symbolic fault injection.

Since most planning approaches as well as our symbolic execution rely on an accurate formal model, we also devised two approaches to aid in the design of formal models. Our first approach learns SMT($\mathcal{LRA}$) formulae from a set of examples and is able to scale significantly better than the state-of-the-art. The second approach uses our symbolic execution engine SEECER to find discrepancies between a formal model and a simulation engine.

The major disadvantage of formal approaches is their poor scalability. Therefore, we also used coverage-guided fuzzing, a test-based method, to uncover errors in the robotic plan. Additionally, we presented a novel coverage metric for the domain of plan-based robotics.

All of our presented approaches were experimentally evaluated. The results support their applicability and utility.

This thesis includes major advances in the area of modelling, verification and test for cognition-enabled robotic plans. Nonetheless, there are still several open research questions to further advance the field, some of which were also discussed in this thesis.

BIBLIOGRAPHY

[1] Iina Aaltonen, Anne Arvola, Päivi Heikkilä, and Hanna Lammi. "Hello Pepper, May I Tickle You? Children's and Adults' Responses to an Entertainment Robot at a Shopping Mall". In: *International Conference on Human-Robot Interaction (HRI)*. 2017, pp. 53–54.

[2] Seyed Aftabjahani and Zainalabedin Navabi. "Functional fault simulation of VHDL gate level models". In: *VHDL International Users Forum Fall Conference (VIUF)*. 1997, pp. 18–23.

[3] Fahad Alaieri and André Vellino. "Ethical Decision Making in Robots: Autonomy, Trust and Responsibility". In: *International Journal of Social Robotics*. 2016, pp. 159–168.

[4] Vítor Alcácer and Virgilio Cruz-Machado. "Scanning the Industry 4.0: A Literature Review on Technologies for Manufacturing Systems". In: *Engineering Science and Technology, an International Journal* (2019), pp. 899–919.

[5] Homa Alemzadeh, Ravishankar Iyer, Zbigniew Kalbarczyk, Nancy Leveson, and Jaishankar Raman. "Adverse Events in Robotic Surgery: A Retrospective Study of 14 Years of FDA Data". In: *PLOS ONE* (2015), pp. 1–20.

[6] Rob Alexander, Heather Rebecca Hawkins, and Andrew John Rae. *Situation coverage–a coverage criterion for testing autonomous robots*. Tech. rep. 2015.

[7] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2017, pp. 319–336.

[8] Saswat Anand, Corina Păsăreanu, and Willem Visser. "Symbolic Execution with Abstract Subsumption Checking". In: *Model Checking Software*. 2006, pp. 163–181.

[9] Dejanira Araiza-Illan, David Western, Anthony Pipe, and Kerstin Eder. "Coverage-Driven Verification — An Approach to Verify Code for Robots that Directly Interact with Humans." In: *Haifa Verification Conference (HVC)*. 2015, pp. 69–84.

[10] Franz Baader and Benjamin Zarrieß. "Verification of Golog programs over description logic actions". In: *International Symposium on Frontiers of Combining Systems (FroCoS)*. 2013, pp. 181–196.

[11] Roberto Baldoni, Emilio Coppa, Daniele D'elia, Camil Demetrescu, and Irene Finocchi. "A survey of symbolic execution techniques". In: *ACM Computing Surveys (CSUR)* (2018), pp. 1–39.

[12] Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. "Extending enumerative function synthesis via SMT-driven classification". In: *Formal Methods in Computer Aided Design (FMCAD)*. 2019, pp. 212–220.

[13] Woodrow Barfield. "Liability for Autonomous and Artificially Intelligent Robots". In: *Paladyn, Journal of Behavioral Robotics* (2018), pp. 193–203.

[14] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. "CRAM — A Cognitive Robot Abstract Machine for everyday manipulation in human environments". In: *International Conference on Intelligent Robots and Systems (IROS)*. 2010, pp. 1012–1017.

[15] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. "A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems". In: *European Conference on Machine Learning (ECML)*. 2005, pp. 23–34.

[16] Robert Bogue. "Underwater robots: a review of technologies and applications". In: *Industrial Robot* (2015), pp. 186–191.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing". In: *Conference on Computer and Communications Security (CCS)*. 2017, pp. 2329–2344.

[18] Niklas Bruns, Vladimir Herdt, and Rolf Drechsler. "Unified HW/SW Coverage: A Novel Metric to Boost Coverage-guided Fuzzing for Virtual Prototype based HW/SW Co-Verification". In: *Forum on Specification & Design Languages (FDL)*. 2022, pp. 1–8.

[19] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. "Symbolic Execution for Software Testing in Practice: Preliminary Assessment". In: *International Conference on Software Engineering (ICSE)*. 2011, pp. 1066–1071.

[20] Balasubramaniyan Chandrasekaran and James Conrad. "Human-robot collaboration: A survey". In: *SoutheastCon*. 2015, pp. 1–8.

[21] Ioan Chisalita, Nahid Shahmehri, and Patrick Lambrix. "Traffic accidents modeling and analysis using temporal reasoning". In: *Conference on Intelligent Transportation Systems (ITSC)*. 2004, pp. 378–383.

[22] Jens Claßen and Gerhard Lakemeyer. "On the Verification of Very Expressive Temporal Properties of Non-terminating Golog Programs". In: *European Conference on Artificial Intelligence (ECAI)*. 2010, pp. 887–892.

[23] Erwin Coumans. *Bullet 2.83 Physics SDK Manual*. 2015. URL: https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/Bullet_User_Manual.pdf (visited on 12/06/2022).

[24] John Danaher. “Robots, law and the retribution gap”. In: *Ethics and Information Technology* (2016), pp. 299–309.

[25] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Available at https://github.com/Z3Prover/z3. 2008, pp. 337–340.

[26] Rodrigo Delgado, Miguel Campusano, and Alexandre Bergel. “Fuzz Testing in Behavior-Based Robotics”. In: *International Conference on Robotics and Automation (ICRA)*. 2021, pp. 9375–9381.

[27] Jeffrey Delmerico et al. “The current state and future outlook of rescue robotics”. In: *Journal of Field Robotics* (2019), pp. 1171–1191.

[28] Louise Dennis, Michael Fisher, Nicholas Lincoln, Alexei Lisitsa, and Sandor Veres. “Practical Verification of Decision-Making in Agent-Based Autonomous Systems”. In: *International Conference on Automated Software Engineering (ASE)* (2013), pp. 305–359.

[29] Alexander Ferrein and Gerhard Lakemeyer. “Logic-based robot control in highly dynamic domains”. In: *Robotics and Autonomous Systems* (2008), pp. 980–991.

[30] Richard Fikes and Nils Nilsson. “Strips: A new approach to the application of theorem proving to problem solving”. In: *Artificial Intelligence* (1971), pp. 189–208.

[31] Andrea Fioraldi, Daniele D’Elia, and Leonardo Querzoni. “Fuzzing Binaries for Memory Safety Errors with QASan”. In: *Secure Development (SecDev)*. 2020, pp. 23–30.

[32] David Fischinger et al. “Hobbit, a care robot supporting independent living at home: First prototype and lessons learned”. In: *Robotics and Autonomous Systems* (2016), pp. 60–78.

[33] Giuseppe Fragapane, René de Koster, Fabio Sgarbossa, and Jan Strandhagen. “Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda”. In: *European Journal of Operational Research* (2021), pp. 405–426.

[34] Yang Gao and Steve Chien. “Review on space robotics: Toward top-level science through space exploration”. In: *Science Robotics* (2017), pp. 1–11.

[35] Michael Gelfond and Vladimir Lifschitz. “Representing action and change by logic programs”. In: *The Journal of Logic Programming* (1993), pp. 301–321.

[36] Malik Ghallab et al. *PDDL - The Planning Domain Definition Language*. Tech. rep. 1998.

[37] Serge Gorbunov and Arnold Rosenbloom. “AutoFuzz: Automated Network Protocol Fuzzing Framework”. In: *International Journal of Computer Science and Network Security (IJCSNS)*. 2012, pp. 239–245.

[38] Henrik Grosskreutz and Gerhard Lakemeyer. “cc-Golog – An Action Language with Continuous Change”. In: *Logic Journal of the IGPL* (2003), pp. 179–221.

[39] Bruno Haible, Michael Stoll, and Sam Steingold. *Implementation Notes for GNU CLISP*. 2010. URL: https://clisp.sourceforge.io/impnotes.html (visited on 12/01/2022).

[40] Tamás Haidegger. “Autonomy for Surgical Robots: Concepts and Paradigms”. In: *Transactions on Medical Robotics and Bionics* (2019), pp. 65–76.

[41] Pavel Hamet and Johanne Tremblay. “Artificial intelligence in medicine”. In: *Metabolism* (2017), S36–S40.

[42] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Roşu. “ROSRV: Runtime Verification for Robots”. In: *International Conference on Runtime Verification (RV)*. 2014, pp. 247–254.

[43] François Ingrand and Malik Ghallab. “Deliberation for autonomous robots: A survey”. In: *Artificial Intelligence* (2014), pp. 10–44.

[44] Özgür Kafali, Alfonso Romero, and Kostas Stathis. “Agent-oriented activity recognition in the event calculus: An application for diabetic patients”. In: *Computational Intelligence* (2017), pp. 899–925.

[45] Mohd Kamarul Bahrin, Fauzi Othman, Nor Hayati Nor Azli, and Muhamad Talib. “Industry 4.0: a Review on Industrial Automation and Robotic”. In: *Jurnal Teknologi* (2016), pp. 137–143.

[46] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. “Grounding and solving in answer set programming”. In: *AI Magazine* (2016), pp. 25–32.

[47] Gayane Kazhoyan, Simon Stelter, Franklin Kenfack, Sebastian Koralewski, and Michael Beetz. “The Robot Household Marathon Experiment”. In: *International Conference on Robotics and Automation (ICRA)*. 2021, pp. 9382–9388.

[48] Hyungsub Kim, Muslum Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu. “PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles”. In: *Network and Distributed System Security Symposium (NDSS)*. 2021.

[49] James King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM (CACM)* (1976), pp. 385–394.

[50] Samuel Kolb, Sergey Paramonov, Tias Guns, and Luc De Raedt. “Learning constraints in spreadsheets and tabular data”. In: *Machine Learning* (2017), pp. 1441–1468.

[51] Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. “Learning SMT(LRA) Constraints Using SMT Solvers”. In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2018, pp. 2333–2340.

[52] Maha Kooli, Alberto Bosio, Pascal Benoit, and Lionel Torres. "Software testing and software fault injection". In: *International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2015, pp. 1–6.

[53] Robert Kowalski and Marek Sergot. "A logic-based calculus of events". In: *New Generation Computing*. 1986, pp. 67–95.

[54] Joost de Kruijff and Hans Weigand. "Formalising Commitments Using the Event Calculus". In: *International Workshop on Value Modelling and Business Ontologies (VMBO)*. 2020, pp. 179–190.

[55] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient State Merging in Symbolic Execution". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2012, pp. 193–204.

[56] Daniel Larsson and Reiner Hähnle. "Symbolic fault injection". In: *International Verification Workshop (VERIFY)*. 2007, pp. 85–103.

[57] Hector Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. "GOLOG: A logic programming language for dynamic domains". In: *The Journal of Logic Programming* (1997), pp. 59–83.

[58] Sam Levin and Julie Wong. "Self-driving Uber kills Arizona woman in first fatal crash involving pedestrian". In: *The Guardian* (2018). URL: https://www.theguardian.com/technology/2018/mar/19/uber-self-driving-car-kills-woman-arizona-tempe (visited on 11/15/2022).

[59] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* (2018), pp. 1–13.

[60] Thomas Lienert, Ludwig Stigler, and Johannes Fottner. "Failure-Handling Strategies For Mobile Robots In Automated Warehouses". In: *European Conference on Modelling and Simulation (ECMS)*. 2019.

[61] Jinguo Liu, Yuechao Wang, Bin Li, and Shugen Ma. "Current Research, Key Performances and Future Development of Search and Rescue Robot". In: *Chinese Journal of Mechanical Engineering (CJME)* (2006), pp. 404–416.

[62] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. "A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution". In: *Haifa Verification Conference (HVC)*. 2014, pp. 284–299.

[63] llvm. *libFuzzer – a library for coverage-guided fuzz testing*. 2022. URL: https://llvm.org/docs/LibFuzzer.html (visited on 10/06/2022).

[64] Matt Luckcuck, Marie Farrell, Louise Dennis, Clare Dixon, and Michael Fisher. “Formal Specification and Verification of Autonomous Robotic Systems: A Survey”. In: *ACM Computing Surveys (CSUR)* (2019), pp. 1–41.

[65] Oded Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook, 2nd ed*. Springer, 2010.

[66] Raphael Mannadiar and Hans Vangheluwe. “Debugging in Domain-Specific Modelling”. In: *International Conference on Software Language Engineering (SLE)*. 2011, pp. 276–285.

[67] Joao Marques-Silva, Inês Lynce, and Sharad Malik. “Conflict-driven clause learning SAT solvers”. In: *Handbook of Satisfiability*. 2021, pp. 133–182.

[68] John McCarthy and Patrick Hayes. “Some Philosophical Problems from the Standpoint of Artificial Intelligence”. In: *Machine Intelligence*. 1969, pp. 463–502.

[69] Drew Mcdermott. “A Reactive Plan Language”. PhD thesis. 1993.

[70] Martin Michalowski, Craig Knoblock, Ken Bayer, and Berthe Choueiry. “Exploiting Automatically Inferred Constraint-Models for Building Identification in Satellite Imagery”. In: *International Symposium on Advances in Geographic Information Systems (GIS)*. 2007.

[71] Barton Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Communications of the ACM (CACM)* (1990), pp. 32–44.

[72] Rob Miller and Murray Shanahan. “Some Alternative Formulations of the Event Calculus”. In: *Computational Logic: Logic Programming and Beyond*. 2002, pp. 452–490.

[73] Leora Morgenstern. “Mid-Sized Axiomatizations of Commonsense Problems: A Case Study in Egg Cracking”. In: *Studia Logica* (2001), pp. 333–384.

[74] Lorenz Mösenlechner. “The Cognitive Robot Abstract Machine”. PhD thesis. Technische Universität München, 2016.

[75] Lorenz Mösenlechner and Michael Beetz. “Fast temporal projection using accurate physics-based geometric reasoning”. In: *International Conference on Robotics and Automation (ICRA)*. 2013, pp. 1821–1827.

[76] Erik Mueller. “Event Calculus Reasoning Through Satisfiability”. In: *Journal of Logic and Computation* (2004), pp. 703–730.

[77] Stephen Muggleton and Luc de Raedt. “Inductive Logic Programming: Theory and Methods”. In: *The Journal of Logic Programming* (1994), pp. 629–679.

[78] Robin Murphy and Dave Hershberger. “Handling Sensing Failures in Autonomous Mobile Robots”. In: *The International Journal of Robotics Research* (1999), pp. 382–400.

[79] Matthew O'Brien, Ronald Arkin, Dagan Harrington, Damian Lyons, and Shu Jiang. "Automatic Verification of Autonomous Robot Missions". In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. 2014, pp. 462–473.

[80] Shashank Pathak, Luca Pulina, Giorgio Metta, and Armando Tacchella. "Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the iCub". In: *International Conference on Intelligent Robots and Systems (IROS)*. 2013, pp. 170–175.

[81] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. "SymPLFIED: Symbolic program-level fault injection and error detection framework". In: *International Conference on Dependable Systems and Networks (DSN)*. 2008, pp. 472–481.

[82] Mikkel Pedersen, Lazaros Nalpantidis, Rasmus Andersen, Casper Schou, Simon Bøgh, Volker Krüger, and Ole Madsen. "Robot skills for manufacturing: From concept to industrial deployment". In: *Robotics and Computer-Integrated Manufacturing* (2016), pp. 282–291.

[83] Edwin Pednault. "ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus". In: *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1989, pp. 324–332.

[84] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* (2011), pp. 2825–2830.

[85] Yvan Petillot, Gianluca Antonelli, Giuseppe Casalino, and Fausto Ferreira. "Underwater Robots: From Remotely Operated Vehicles to Intervention-Autonomous Underwater Vehicles". In: *IEEE Robotics & Automation Magazine* (2019), pp. 94–101.

[86] Mukul Prasad, Armin Biere, and Aarti Gupta. "A survey of recent advances in SAT-based formal verification". In: *International Journal on Software Tools for Technology Transfer (STTT)* (2005), pp. 156–173.

[87] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing". In: *Network and Distributed System Security Symposium (NDSS)*. 2017, pp. 1–14.

[88] Raymond Reiter. "The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression." In: *Artificial and Mathematical Theory of Computation*. 1991, pp. 359–380.

[89] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. "cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis". In: *International Conference on Computer Aided Verification (CAV)*. 2019, pp. 74–83.

[90] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, 2009.

[91] Koushik Sen. "Concolic testing". In: *International Conference on Automated Software Engineering (ASE)*. 2007, pp. 571–572.

[92] Murray Shanahan. "A circumscriptive calculus of events". In: *Artificial Intelligence* (1995), pp. 249–284.

[93] Murray Shanahan. "Robotics and the Common Sense Informatic Situation". In: *European Conference on Artificial Intelligence (ECAI)*. 1996, pp. 684–688.

[94] Murray Shanahan. "An abductive event calculus planner". In: *The Journal of Logic Programming* (2000), pp. 207–240.

[95] Murray Shanahan. "An attempt to formalise a non-trivial benchmark problem in common sense reasoning". In: *Artificial Intelligence* (2004), pp. 141–165.

[96] Robin Sibson. "SLINK: An Optimally Efficient Algorithm for the Single-Link Cluster Method". In: *The Computer Journal* (1973), pp. 30–34.

[97] Reid Simmons and David Apfelbaum. "A task description language for robot control". In: *International Conference on Intelligent Robots and Systems (IROS)*. 1998, pp. 1931–1937.

[98] Reid Simmons, Charles Pecheur, and Grama Srinivasan. "Towards automatic verification of autonomous systems". In: *International Conference on Intelligent Robots and Systems (IROS)*. 2000, pp. 1410–1415.

[99] Fernando Soto, Jie Wang, Rajib Ahmed, and Utkan Demirci. "Medical Micro/Nanorobots in Precision Medicine". In: *Advanced Science* (2020).

[100] Bogdan-Andrei Tabacaru, Moomen Chaari, Wolfgang Ecker, Thomas Kruse, and Cristiano Novello. "Fault-effect analysis on system-level hardware modeling using virtual prototypes". In: *Forum on Specification and Design Languages (FDL)*. 2016, pp. 1–7.

[101] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew Walter, Ashis Banerjee, Seth Teller, and Nicholas Roy. "Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation". In: *AAAI Conference on Artificial Intelligence*. 2011, pp. 1507–1514.

[102] Moritz Tenorth, Daniel Nyga, and Michael Beetz. "Understanding and executing instructions for everyday manipulation tasks from the World Wide Web". In: *IEEE International Conference on Robotics and Automation*. 2010, pp. 1486–1491.

[103] Pradib Thaker, Vishwani Agrawal, and Mona Zaghloul. "Register-transfer level fault modeling and test evaluation techniques for VLSI circuits". In: *International Test Conference (ITC)*. 2000, pp. 940–949.

[104] Michael Thielscher. “Ramification and causality”. In: *Artificial Intelligence* (1997), pp. 317–364.

[105] Michael Thielscher. *The fluent calculus*. Tech. rep. 2000.

[106] Madhura Thosar, Sebastian Zug, Alpha Skaria, and Akshay Jain. “A Review of Knowledge Bases for Service Robots in Household Environments”. In: *International Workshop on Artificial Intelligence and Cognition (AIC)*. 2018, pp. 940–949.

[107] Abhishek Udupa, Arun Raghavan, Jyotirmoy Deshmukh, Sela Mador-Haim, Milo Martin, and Rajeev Alur. “TRANSIT: specifying protocols with concolic snippets”. In: *ACM SIGPLAN Notices* (2013), pp. 287–296.

[108] Saurabh Vaidya, Prashant Ambad, and Santosh Bhosle. “Industry 4.0 – A Glimpse”. In: *Procedia Manufacturing* (2018), pp. 233–238.

[109] Leslie Valiant. “A Theory of the Learnable”. In: *Communications of the ACM (CACM)* (1984), pp. 1134–1142.

[110] Tijs Vandemeulebroucke, Bernadette Dierckx de Casterlé, and Chris Gastmans. “The use of care robots in aged care: A systematic review of argument-based ethics literature”. In: *Archives of Gerontology and Geriatrics* (2018), pp. 15–25.

[111] Alessandro Vercelli, Innocenzo Rainero, Ludovico Ciferri, Marina Boido, and Fabrizio Pirri. “Robots in Elderly Care”. In: *Scientific Journal on Digital Cultures (DigitCult)* (2018), pp. 37–50.

[112] Vandi Verma et al. “First 210 solar days of Mars 2020 Perseverance Robotic Operations - Mobility, Robotic Arm, Sampling, and Helicopter”. In: *Aerospace Conference (AeroConf)*. 2022, pp. 1–20.

[113] Lihui Wang, Sichao Liu, Hongyi Liu, and Xi Wang. “Overview of Human-Robot Collaboration in Manufacturing”. In: *International Conference on the Industry 4.0 Model for Advanced Manufacturing (AMP)*. 2020, pp. 15–58.

[114] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Koay, Kerstin Dautenhahn, and Joan Saez-Pons. “Toward Reliable Autonomous Robotic Assistants Through Formal Verification: A Case Study”. In: *Transactions on Human-Machine Systems* (2016), pp. 186–196.

[115] Georgia Wells. “Security Robot Suspended After Colliding With a Toddler”. In: *The Wall Street Journal* (2016). URL: https://www.wsj.com/articles/security-robot-suspended-after-colliding-with-a-toddler-1468446311 (visited on 11/15/2022).

[116] Alan Winfield, Katie Winkle, Helena Webb, Ulrik Lyngs, Marina Jirotka, and Carl Macrae. “Robot Accident Investigation: A Case Study in Responsible Robotics”. In: *Software Engineering for Robotics*. 2021, pp. 165–187.

[117] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. “Fuzzing Mobile Robot Environments for Fast Automated Crash Detection”. In: *International Conference on Robotics and Automation (ICRA)*. 2021, pp. 5417–5423.

[118] Aimee van Wynsberghe. “Designing Robots for Care: Care Centered Value-Sensitive Design”. In: *Science and Engineering Ethics* (2013), pp. 407–433.

[119] Georgios Zachiotis, George Andrikopoulos, Randy Gornez, Keisuke Nakamura, and George Nikolakopoulos. “A Survey on the Application Trends of Home Service Robotics”. In: *International Conference on Robotics and Biomimetics (ROBIO)*. 2018, pp. 1999–2006.

[120] Michal Zalewski. *Technical "whitepaper" for afl-fuzz*. 2017. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited on 10/06/2022).

,