Verifying Safety Properties of Robotic Plans operating in Real-World Environments via Logic-based Environment Modeling*

Tim Meywerk[1][0000−0002−5960−5456], Marcel Walter[1][0000−0001−5660−9518], Vladimir Herdt[2][0000−0002−4481−057X], Jan Kleinekathöfer[1,2][0000−0001−6357−0914], Daniel Große[2,3][0000−0002−1490−6175], and Rolf Drechsler[1,2][0000−0002−9872−1740]

[1] Research Group of Computer Architecture, University of Bremen, Germany
[2] Cyber Physical Systems, DFKI GmbH, Bremen, Germany
[3] Chair of Complex Systems, Johannes Kepler University Linz, Austria
{tmeywerk,m_walter,vherdt,ja_kl,drechsler}@uni-bremen.de
daniel.grosse@jku.at

Abstract. These days, robotic agents are finding their way into the personal environment of many people. With robotic vacuum cleaners commercially available already, comprehensive cognition-enabled agents assisting around the house autonomously are a highly relevant research topic. To execute these kinds of tasks in constantly changing environments, complex goal-driven control programs, so-called *plans*, are required. They incorporate perception, manipulation, and navigation capabilities among others. As with all technological innovation, consequently, safety and correctness concerns arise.

In this paper, we present a methodology for the verification of safety properties of robotic plans in household environments by a combination of environment reasoning using *Discrete Event Calculus* (DEC) and *Symbolic Execution* for effectively handling symbolic input variables (e. g. object positions). We demonstrate the applicability of our approach in an experimental evaluation by verifying safety properties of robotic plans controlling a two-armed, human-sized household robot packing and unpacking a shelf. Our experiments demonstrate our approach's capability to verify several robotic plans in a realistic, logically formalized environment.

Keywords: Cognition-enabled Robotics · Household Robots · Formal Verification · Symbolic Execution · Discrete Event Calculus.

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

1 Introduction

These days, robotic agents are finding their way into the personal environment of many people; for example in the form of autonomous vacuum cleaner robots. Ambitious research is conducted in the direction of fully autonomous household robots solving complex tasks like tea serving [20] or cooking [1]. In contrast to their ancestors—industrial robots that were only utilized for repetitive and physically strenuous work—these household robots operate in highly complex, constantly changing environments. To achieve their goals, more than a simple pre-programmed action sequence is required to control them. There is a need for cognitive mechanisms that allow robotic agents to interact with their environments based on the execution of general tasks. These include, but are not limited to, reasoning about spatial relations of objects and, based on that, deciding which action leads to the intended environment manipulation. Approaches based on cognitive mechanisms have proven their usefulness and, as a consequence, learning, knowledge processing, and action planning found entry into robot control programs, which are usually called (robotic) *plans*. For programming plans, many high-level *planning languages* have been developed. Some examples are RPL [5], RMPL [30], and CPL [2]. These languages combine a rich, Turing-complete semantic with the ability to natively call low-level subroutines like perception, navigation, and manipulation. The *CRAM Plan Language* (CPL) in particular is an extension of the *Common Lisp* programming language and part of the *Cognitive Robot Abstract Machine* (CRAM) toolbox [2]. CRAM provides a multitude of environment interaction and reasoning modules.

With such complex software systems and challenging tasks in real-world household environments, reliability is more important than ever. Since simulation and testing quickly reach their limits in guaranteeing this reliability, formal safety verification methods become indispensable. Recently, in [16], we have proposed a procedure based on *Symbolic Execution* [10,3] for the verification of CPL code that we called *SEECER*. As shown in that paper, reasoning about the robotic plan by itself offers only limited benefits. Instead, the plan must be related to its intended environment. In particular, interaction between the robot and the environment needs to be taken into account to produce comprehensive verification results. In [16], the relatively abstract *Wumpus World* [24] has been used as an environment model. To allow easier integration with the CPL, the environment was modeled directly in the Common Lisp programming language. However, environment models in Common Lisp have to be specifically adjusted to the plan under verification. Therefore, each plan requires its own environment model, which can be used for no other purpose than the plan verification. Yet, there are several logical formalisms specialized in the modeling of environments and actions, for example, *Situation Calculus* and its predecessors [14] and *(Discrete) Event Calculus* [11,26,17,19]. They are regularly used to model household environments and robotic actions [25,18]. These formalisms have several advantages over a model in Common Lisp, such as their well-defined semantics and a plethora of environment descriptions and reasoning procedures proposed in the literature.

In this paper, we propose a safety verification methodology of robotic plans written in a high-level planning language—we use CPL as a running example—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 that we combine with symbolic execution. Our contribution in this paper 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 is achieved via a reduction to a pure DEC reasoning problem. To the best of our knowledge, such a reduction has not been proposed before. Additionally, our procedure serves as an important building block for our second and major contribution, namely the verification of more complex robotic plans in combination with a DEC environment model and symbolic execution. Our third contribution is the verification of several plans (that are taken from the CRAM repository) in a detailed household environment and the modeling of this very environment in DEC. Our experiments demonstrate our approach's capability to verify several robotic plans in a realistic, logically formalized environment.

The remainder of this paper is structured as follows: Section 2 reviews preliminaries necessary to keep this work self-contained. Section 3 introduces our safety verification methodology. In Section 4, we evaluate the applicability of our approach by verifying safety properties of CPL plans controlling a two-armed, human-sized robot packing and unpacking a shelf. Section 5 concludes the paper.

2 Preliminaries

In this section, we discuss all the necessary preliminaries to keep this paper self-contained. We give an overview of the CPL in Section 2.1 first, and discuss our recent approach for symbolic execution of the CPL in Section 2.2. Afterwards, we introduce DEC in Section 2.3.

2.1 CRAM Plan Language

Many AI and robotics systems utilize the *Lisp* programming language to this day because libraries and frameworks written in or at least mainly supporting Lisp are available. The *Cognitive Robot Abstract Machine* (CRAM) [2] is a prominent example as it is written in *Common Lisp*, a dialect of the Lisp programming language. CRAM provides an interface for perception modules, belief states, knowledge bases, and navigation and manipulation actuators to be used in robotic systems. Additionally, it exposes the *CRAM Plan Language* (CPL) which allows writing high-level plan descriptions in Common Lisp. An abstract overview of the architecture of the CRAM stack can be seen in Figure 1. The execution is controlled by the CPL plan, which interacts with the environment through perception, manipulation, and navigation subroutines. To reason about the next action to be taken, a belief state and knowledgebase can be consulted through a query-answer interface.

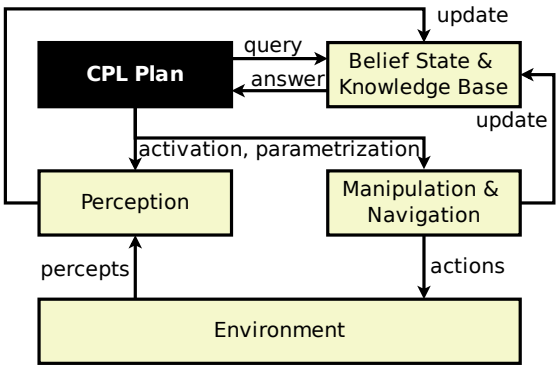

Fig. 1: Overview of the CRAM stack architecture

In CPL, plans describe desired behavior in terms of hierarchies of goals, rather than fixed sequences of actions that need to be performed. All exchange of information with parts of the CRAM stack, e. g., the perception modules or knowledge bases, as well as interaction with actuators to manipulate the environment, works via *designators* which are a common concept among reasoning and planning systems. Designators thereby often encapsulate high-level descriptions that are familiar to humans but abstract to robots like a *location* or a *motion*. Classes of designators available in CPL are for instance

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

In CPL, an action designator contains the action type to perform (like detecting or grasping) and several parameters. It can be passed to the `perform` function, which breaks it down to sub-tasks and takes care of their execution. Both, action designators and the `perform` function, are particularly important for this work and will be utilized in Section 3. The following example illustrates a typical use of different designators.

Example 1. Figure 2 shows a typical snippet taken from a CPL plan that uses multiple different designators to indicate action, motion, and location. In CPL, the designators are generated using the `a` and `an` macros. The plan snippet performs a motion to turn the robot's head to look at a specified target position and places the robot's arm to the same location in parallel (indicated by the `par` function). As shown here, designators may be nested, i. e., contain other designators such as the location designators contained within an action and a motion designator respectively.

```
1 (defun place-object (?target-pose ?arm)
2   (par
3     (perform (a motion (type looking)
4         (target (a location (pose ?target-pose)))))
5     (perform (an action (type placing)
6         (arm ?arm) (target (a location (pose ?target-pose)))))))
```

Fig. 2: Designator usage in CPL code

In the following section, we review our recent approach to verify properties of CPL plans.

2.2 Symbolic Execution for the CRAM Plan Language

In [16], we utilized a technique known as *Symbolic Execution* [10,3] to verify annotated assertions on plans written in CPL. Due to the high abstraction level, the CPL plans were first translated to an Assembly-like intermediate language that we called *Intermediate Plan Verification Language* (IPVL). The CPL-to-IPVL compiler that we implemented also integrated an environment model that had to be written in Common Lisp as well. This model was then also compiled to IPVL code. Safety properties were also modeled in Common Lisp in the form of `assert` instructions. All designators included in the plan were mocked by the environment model to abstract from underlying sensor and actuator operations. To handle this IPVL code, we implemented the symbolic interpreter *SEECER* (Symbolic Execution Engine for Cognition-Enabled Robotics).

SEECER analyzes the plan path-wise while managing a set of *symbolic execution states*. A symbolic execution state is a 3-tuple (pc, ip, α). Here, pc is the *path condition*, which encapsulates all restrictions that are imposed on the execution state; ip is the *instruction pointer*, which points to the next instruction to be executed; and α is the *variable map*, which maps plan variables to their symbolic value. Non-control flow instructions (e. g. arithmetic instructions) update α by changing the target variable to its new value and increment ip by 1. Branching instructions of the form `if C goto i` with a conditional C and instruction i are evaluated as follows: the feasibility of both branches is checked using an SMT solver, i. e. the formulas $pc \wedge C$ and $pc \wedge \neg C$ are checked for satisfiability. If only one of the formulas is satisfiable, the respective branch is taken and ip is updated accordingly. If both formulas are satisfiable however, the execution state is duplicated. One copy follows the jump, i. e. the path condition pc is updated to $pc \wedge C$ and ip is set to i. The other copy appends $\neg C$ to the path condition and resumes with the next instruction. Whenever SEECER encounters an `assert(a)` instruction, the satisfiability of $pc \wedge \neg a$ is evaluated. A satisfying assignment corresponds to an error in the plan. If the symbolic execution terminates without finding such an error, the plan is proven to be safe.

Finally, we conducted a case study on the rather simplistic *Wumpus World* [24] environment that we modeled in Common Lisp to demonstrate the general feasibility of our approach.

While this paper uses some concepts from our previous work [16], we focus on more real-world scenarios in this paper. Furthermore, as an environment model, we rely on a logic formalism called *Discrete Event Calculus* (DEC) due to its high expressiveness and simultaneous decidability. Since we use DEC in this paper for modeling environments as well as verifying action sequences and plans, we give an overview on DEC in the following section.

2.3 Discrete Event Calculus

The *event calculus* [11,26,17] is an established formalism to model and reason about events and their consequences. It allows for the modeling of non-determinism, conditional effects of events, state constraints, and gradual change, among others. A domain description modeled in the event calculus follows the *commonsense 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 modeler may however choose to *release* certain properties from this law. Furthermore, the event calculus allows us 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 [27], traffic accidents [4], diabetic patients [8] and smart contracts [12].

In [19] 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]

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 modeling 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 paper, we will focus mostly on actions and will, therefore, use event and action synonymously. 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 $going(location)$ or the fluent $isAt(object, location)$.

[4] Gradual change allows to model properties that change over time after an initial action, e. g., an object falling and eventually hitting the ground after it has been dropped.

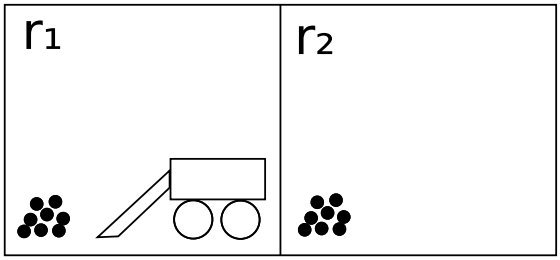

Fig. 3: Visualization of the vacuum world ($n = 2$)

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:

- *Happens*(a, t): Action a happens at timepoint t.
- *HoldsAt*(f, t): Fluent f is true at timepoint t.
- *ReleasedAt*(f, t): Fluent f is released from the commonsense law of inertia at timepoint t.
- *Initiates*(a, f, t): When action a happens at timepoint t, then fluent f will be true at timepoint $t + 1$.
- *Terminates*(a, f, t): When action a happens at timepoint t, then fluent f will be false at timepoint $t + 1$.
- *Releases*(a, f, t): When action a happens at timepoint t, then fluent f will be released from the commonsense 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 2. Consider the modeling of a simple robotic environment inspired by the *vacuum world* [23]. 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_{i+1} 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 Figure 3. 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 *Left*, *Right* and *Clean* and the fluents *RobotInRoom*(*room*) and *Dirty*(*room*).

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

$$\begin{aligned}\forall t : \exists r : &\big(\mathit{HoldsAt}(\mathit{RobotInRoom}(r), t)\big)\\ \forall t, r_i, r_j : &\big(\mathit{HoldsAt}(\mathit{RobotInRoom}(r_i), t) \wedge \mathit{HoldsAt}(\mathit{RobotInRoom}(r_j), t) \implies\\ &r_i = r_j\big)\end{aligned}$$

After that we describe the effects of the robot's actions. The *Left* and *Right* 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:

$$\begin{aligned}\forall t, r : &\big(\mathit{HoldsAt}(\mathit{RobotInRoom}(r), t) \wedge r \neq r_1 \implies\\ &\mathit{Initiates}(\mathit{Left}, \mathit{RobotInRoom}(r - 1), t) \wedge\\ &\mathit{Terminates}(\mathit{Left}, \mathit{RobotInRoom}(r), t)\big)\\ \forall t, r : &\big(\mathit{HoldsAt}(\mathit{RobotInRoom}(r), t) \wedge r \neq r_n \implies\\ &\mathit{Initiates}(\mathit{Right}, \mathit{RobotInRoom}(r + 1), t) \wedge\\ &\mathit{Terminates}(\mathit{Right}, \mathit{RobotInRoom}(r), t)\big)\end{aligned}$$

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

$$\forall t, r : \big(\mathit{HoldsAt}(\mathit{RobotInRoom}(r), t) \implies \mathit{Terminates}(\mathit{Clean}, \mathit{Dirty}(r), t)\big)$$

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

Axioms Following the notation from [19], 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 commonsense 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

$$\begin{aligned}&\big(\mathit{HoldsAt}(f, t) \wedge \neg \mathit{ReleasedAt}(f, t + 1) \wedge\\ &\neg\exists a : \big(\mathit{Happens}(a, t) \wedge \mathit{Terminates}(a, f, t)\big)\big) \implies\\ &\mathit{HoldsAt}(f, t + 1)\end{aligned}$$

AXIOM DEC6

$$\begin{aligned}&\big(\neg\mathit{HoldsAt}(f, t) \wedge \neg \mathit{ReleasedAt}(f, t + 1) \wedge\\ &\neg\exists a : \big(\mathit{Happens}(a, t) \wedge \mathit{Initiates}(a, f, t)\big)\big) \implies\\ &\neg\mathit{HoldsAt}(f, t + 1)\end{aligned}$$

Axiom Dec7
$\big(ReleasedAt(f,t) \wedge$
$\neg\exists a : \big(Happens(a,t) \wedge \big(Initiates(a,f,t) \vee Terminates(a,f,t)\big)\big)\big) \implies$
$ReleasedAt(f,t+1)$

Axiom Dec8
$\big(\neg ReleasedAt(f,t) \wedge$
$\neg\exists a : \big(Happens(a,t) \wedge Releases(a,f,t)\big)\big) \implies$
$\neg ReleasedAt(f,t+1)$

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 commonsense law of inertia. If some action happens that releases a fluent, that fluent will be released at the next timepoint.

Axiom Dec9
$\big(Happens(a,t) \wedge Initiates(a,f,t)\big) \implies HoldsAt(f,t+1)$

Axiom Dec10
$\big(Happens(a,t) \wedge Terminates(a,f,t)\big) \implies \neg HoldsAt(f,t+1)$

Axiom Dec11
$\big(Happens(a,t) \wedge Releases(a,f,t)\big) \implies ReleasedAt(f,t+1)$

Axiom Dec12
$\big(Happens(a,t) \wedge (Initiates(a,f,t) \vee Terminates(a,f,t))\big) \implies \neg Released(f,t+1)$

Let the conjunction of axioms Dec5 to Dec12 be $\mathrm{Ax}_{\mathrm{Dec}}$.

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

Example 3. Consider again the DEC description from Example 2. 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:

$$HoldsAt(RobotInRoom(r_1), 0)$$

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

$$Happens(Right, 0)$$

When combining this extended description with the $\mathrm{Ax}_{\mathrm{Dec}}$ axioms, we can infer $HoldsAt(RobotInRoom(r_2), 1)$ 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 [18,29] or automatically in highly restricted settings [28]. The DEC on the other hand allows for fully automated reasoning by restricting all domains, including the timepoints, to a finite set. 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) [19]. 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.

Comparison to Other Formalisms Over the years, several formalisms for the description of actions and their effects have been proposed. Prominent examples are the action languages $\mathcal{A}$ [6], ADL [21] and PDDL [15] and their extensions. In contrast to DEC, these formalisms have a restricted expressive power which allows for 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 modeled. For instance, non-determinism, ramification constraints, gradual change, or multiple agents can all be expressed in the DEC, but are often problematic for the aforementioned action languages. In the context of this paper, reasoning about an environment is combined with symbolic execution on a Turing-complete planning language. In this scenario, reasoning is only possible over a finite number of timepoints anyway, making the use of a restricted action language unnecessary.

A closer relative of the (discrete) event calculus is the situation calculus [13,22]. The two formalisms are very similar in that they both reason about actions and change. Their differences are rather subtle. The major reason why we choose the DEC over the situation calculus for this work is the ability to easier model the exact time at which an action occurs, including concurrent actions. Even though this ability is not extensively used in this publication, we expect it to prove its usefulness in future works.

3 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

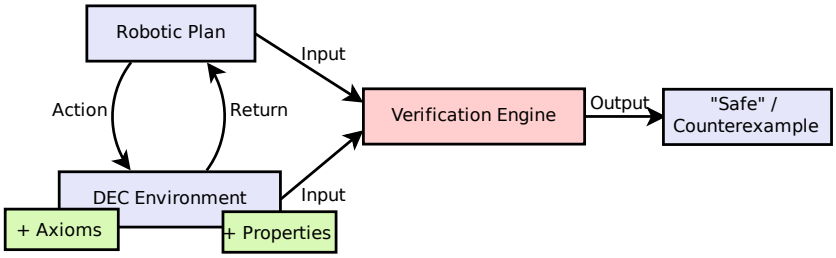

Fig. 4: Abstract view on the considered verification problem

of the considered topics and the structure of this part in the following Section 3.1. In Section 3.2, we first cover the verification of simple action sequences and in Section 3.3, we present our verification approach, which is based on symbolic execution, for complex plans written in CPL.

3.1 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 Figure 4 and intuitively reads as follows: given a robotic plan and a DEC description consisting of an environment description formulated in DEC, 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 do not hold? The approach that we are proposing implements the verification engine shown in Figure 4 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 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 the 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, afterward, present our combined approach for complex plans.

3.2 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 axiomatization $\mathrm{Ax}_{\mathrm{DEC}}$,

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 conjuction of DEC axioms, environment description and action occurrences entails the safety properties, i. e.

$$\mathrm{Ax}_{\mathrm{DEC}} \wedge \mathit{Env} \wedge \mathtt{CIRC}[\bigwedge_{i=1}^{k} \mathit{Happens}(a_i, i-1); \mathit{Happens}] \models \bigwedge_{j=1}^{l} P_j.$$

Here, `CIRC` is the circumscription operator introduced in Section 2.3. In this case, it ensures that only the actions $a_1, \ldots, a_k$ are occurring.

Since most reasoners for DEC, including the DEC reasoner introduced in Section 2.3, 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

$$\begin{aligned}\mathrm{Ax}_{\mathrm{DEC}} \wedge \mathit{Env} \wedge \mathtt{CIRC}[\bigwedge_{i=1}^{k} \mathit{Happens}(a_i, i-1); \mathit{Happens}] \wedge \\ \mathtt{CIRC}[\bigwedge_{j=1}^{l} (\neg P_j \implies U); U] \wedge U,\end{aligned}$$

where U (short for *unsafe*) is a new 0-ary predicate symbol. 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 [19]. 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 4. Consider again the vacuum world with $n = 2$ from the previous examples. Consider further the following action sequence: *Left*, *Clean*, *Right*, *Clean*. Assume that we want to verify that this action sequence results in all rooms being cleaned. We express this by the property $P_1 = \forall r : \big(\neg \mathit{HoldsAt}(\mathit{Dirty}(r), 4)\big)$. The verification is now conducted by model finding on the following conjunction:

$$\begin{aligned}&\mathrm{Ax}_{\mathrm{DEC}} \wedge \mathit{Vac}_2 \wedge \mathtt{CIRC}[\mathit{Happens}(\mathit{Left}, 0) \wedge \mathit{Happens}(\mathit{Clean}, 1) \wedge \\ &\mathit{Happens}(\mathit{Right}, 2) \wedge \mathit{Happens}(\mathit{Clean}, 3); \mathit{Happens}] \wedge \\ &\mathtt{CIRC}[\exists r : \big(\mathit{HoldsAt}(\mathit{Dirty}(r), 4)\big) \implies U; U] \wedge U,\end{aligned}$$

where Vac_2 is the DEC description of the vacuum world described in Example 2 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.

3.3 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

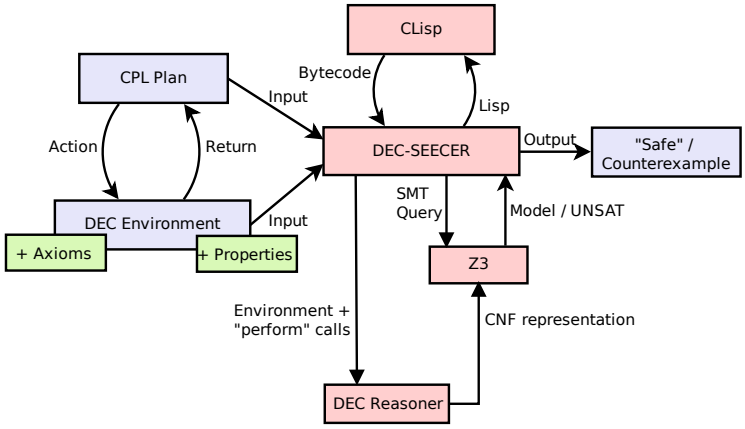

Fig. 5: DEC-centric architectural view

no longer sufficient to solve the verification task for arbitrary plans written in Turing-complete planning 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 planning language, as long as a suitable symbolic execution engine is available.

Figure 5 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, which we have previously presented in [16]. 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 formulas. These formulas can be combined with other SMT constraints and solved by the SMT solver *Z3*. Like in [16], a `perform` mock is used to abstract from low-level effects like motor control. In contrast to our previous work, however, this mock is not hand-written for each environment, but can instead handle arbitrary DEC descriptions, thus supporting a multitude of different environments. To facilitate this more general `perform` mock, we decided to replace the intermediate representation IPVL from [16] with a more general and mature intermediate representation, namely the *CLISP bytecode* generated by the Common Lisp implementation *CLISP* [7].

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 2.2 by managing several execution states. These execution states are however represented by a 4-tuple (pc, ip, α, E), where pc, ip and α are the path condition, instruction pointer and variable map known from Section 2.2, 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 3.2. The DEC description of the initial state is given as

$$E_0 = \mathrm{Ax}_{\mathrm{DEC}} \wedge \mathit{Env} \wedge \texttt{CIRC}[\bigwedge_{j=1}^{l} (\neg P_j \implies U); U]$$

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 3.2, these *Happens*() conjuncts are subject to circumscription. `perform` instructions also increase the instruction pointer ip by 1, but do not affect pc and α. The third type, branching instructions, lead to a feasibility check of both branches. To account for effects 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 $\mathit{DECR}(E)$. 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 $m(E)$. DEC-SEECER now evaluates the satisfiability of both $C \wedge pc \wedge \mathit{DECR}(E) \wedge m(E)$ and $\neg C \wedge pc \wedge \mathit{DECR}(E) \wedge m(E)$. 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:

$$pc \wedge \mathit{DECR}(E) \wedge \mathit{DECR}(U) \wedge m(E)$$

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 5. 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'.

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

Fig. 6: CPL plan for the vacuum world

Assume we want to verify the safety of the CPL plan shown in Figure 6. This plan is more complex than the action sequence presented in Example 4 because it considers the state of the environment in Line 3 and 3 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 4. 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

$$P_2 = \forall t, r : \big(\neg \mathit{HoldsAt}(\mathit{Dirty}(r), t) \wedge \\ \mathit{HoldsAt}(\mathit{RobotInRoom}(r), t) \implies \neg \mathit{Happens}(\mathit{Clean}, t)\big).$$

The initial symbolic execution state can now be written as the 4-tuple $(\mathit{true}, 0, \emptyset, E_0)$ with

$$E_0 = \mathrm{Ax}_{\mathrm{DEC}} \wedge \mathit{Vac'}_2 \wedge \texttt{CIRC}[\exists r : \big(\mathit{HoldsAt}(\mathit{Dirty}(r), t_{max})\big) \implies U \wedge \\ \exists t, r : \big(\neg \mathit{HoldsAt}(\mathit{Dirty}(r), t) \wedge \mathit{HoldsAt}(\mathit{RobotInRoom}(r), t) \wedge \\ \mathit{Happens}(\mathit{Clean}, t)\big) \implies U; U].$$

Figure 7 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 Figure 6), 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:

$$E_1 = E_0 \wedge \texttt{CIRC}[\mathit{Happens}(\mathit{Left}, 0); \mathit{Happens}]$$
$$E_2 = E_0 \wedge \texttt{CIRC}[\mathit{Happens}(\mathit{Left}, 0) \wedge \mathit{Happens}(\mathit{Detect}, 1); \mathit{Happens}]$$
$$\alpha_2 = \{\texttt{dirty} \mapsto \mathit{DECR}(\mathit{HoldsAt}(\mathit{ReturnVal}(), 2)\}$$

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:

$$\alpha_2(\text{dirty}) \wedge \mathit{DECR}(E_2) \wedge \mathit{DECR}(U) = \\ \mathit{DECR}(\mathit{HoldsAt}(\mathit{ReturnVal}(), 2) \wedge \mathit{DECR}(E_2) \wedge \mathit{DECR}(U)$$

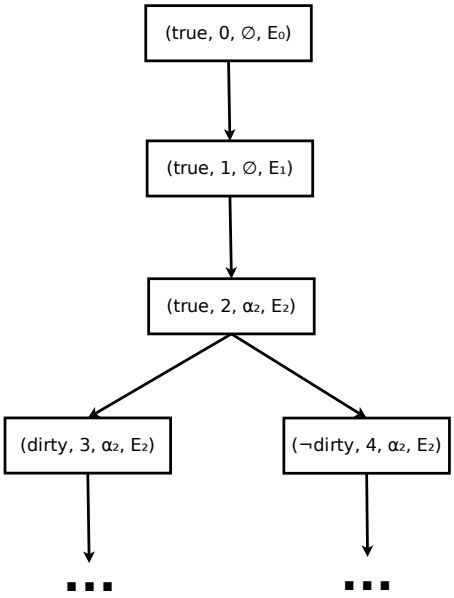

Fig. 7: Execution tree of the symbolic execution

This formula is unsatisfiable. In fact, every such formula during the symbolic execution of this plan is unsatisfiable, thus proving safety of said 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. Here, we evaluate our approach on a real-world scenario.

4 Experimental Evaluation

We implemented DEC-SEECER using the DEC reasoner version 1.0, the SMT solver Z3 version 4.8.4, and CLISP version 2.49.93+ as back-end. To evaluate our approach, we used several variations of the *Shopping Demo* plan taken from the official CRAM repository [2]. 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

```
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)))
```

Fig. 8: Excerpt of the Shopping Demo plan

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 modeled 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 4.1, we discuss all plans in detail. In Section 4.2, we present our DEC environment and the safety properties. Finally, in Section 4.3, we show our experimental results and discuss them.

4.1 Robotic Plans

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

Shopping Demo 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. Afterward, 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 in 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.

Figure 8 shows an excerpt of the Shopping Demo. 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).

Modified Shopping Demo 1 This plan is a modified version of the Shopping Demo. A small error was deliberately inserted to test our approach's bug-finding capabilities. By replacing the `>=` in Line 1 of Figure 8 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.

Modified Shopping Demo 2 This plan is another erroneous modification of the original Shopping Demo, too. 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 Demo plan.

Modified Shelf Filling 1 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.

Modified Shelf Filling 2 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.

All plans presented in this section are evaluated in the DEC environment description, which we explain next.

4.2 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 modeled 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 modeling the position 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[5], 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 Demo, 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.

4.3 Experimental Results

We ran our proposed verification approach on all plans presented in Section 4.1. All Shopping Demo 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. All experiments have been conducted on a Linux machine with an Intel Xeon CPU with 3.5 GHz clock rate.

Table 1 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

[5] 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 1: Verification results

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

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 Demo 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. [9] 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.

5 Conclusion

The verification of safety properties of robotic plans is an indispensable prerequisite for using autonomous robotic agents in everyday scenarios. Although this task is of paramount importance, it is also extremely difficult, since one is dealing with Turing-complete high-level plans operating in constantly changing environments.

In this paper, we presented a methodology that addresses this problem. We were able to verify plans operating in household environments by combining Discrete Event Calculus and symbolic execution. This integration allows for the formal verification of general robotic plans in arbitrary environments modeled in DEC. We exemplary showed, by means of experimental evaluation, that we can verify safety properties of several plans controlling a two-armed human-sized household robot packing and unpacking a shelf.

While we could demonstrate the general applicability of our proposed approach by the means of an experimental evaluation, it also indicated that there is room for performance improvements. In future work, we, therefore, want to rebuild the current DEC reasoner from scratch incorporating state-of-the-art grounding techniques and an incremental approach.

References

1. Beetz, M., Klank, U., Kresse, I., Maldonado, A., Mösenlechner, L., Pangercic, D., Rühr, T., Tenorth, M.: Robotic roommates making pancakes. In: IEEE-RAS International Conference on Humanoid Robots (2011)
2. Beetz, M., Mösenlechner, L., Tenorth, M.: Cram—a cognitive robot abstract machine for everyday manipulation in human environments. In: Intelligent Robots and Systems. pp. 1012–1017. IEEE (2010)
3. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Commun. ACM **56**(2), 82–90 (Feb 2013)
4. Chisalita, I., Shahmehri, N., Lambrix, P.: Traffic accidents modeling and analysis using temporal reasoning. In: Conference on Intelligent Transportation Systems (ITSC). vol. 7, pp. 378–383 (2004)
5. Drabble, B.: EXCALIBUR: a program for planning and reasoning with processes. Artificial Intelligence **62**(1), 1–40 (1993)
6. Gelfond, M., Lifschitz, V.: Representing action and change by logic programs. The Journal of Logic Programming **17**(2), 301 – 321 (1993)
7. Haible, B., Stoll, M., Steingold, S.: Implementation notes for gnu clisp (2010)
8. Kafali, Ö., Romero, A.E., Stathis, K.: Agent-oriented activity recognition in the event calculus: An application for diabetic patients. Computational Intelligence **33**(4), 899–925 (2017)
9. Kaufmann, B., Leone, N., Perri, S., Schaub, T.: Grounding and solving in answer set programming. AI Magazine **37**(3), 25–32 (2016)
10. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (Jul 1976)
11. Kowalski, R., Sergot, M.: A logic-based calculus of events. In: New Generation Computing. vol. 4, pp. 67–95 (1986)

12. de Kruijff, J., Weigand, H.: Formalising commitments using the event calculus. In: VMBO (2020)
13. McCarthy, J.: Situations, actions, and causal laws. Tech. rep. (1963)
14. McCarthy, J., Hayes, P.J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969)
15. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: Pddl—the planning domain definition language (1998)
16. Meywerk, T., Walter, M., Herdt, V., Große, D., Drechsler, R.: Towards Formal Verification of Plans for Cognition-enabled Autonomous Robotic Agents. In: Euromicro Conference on Digital System Design (DSD). pp. 129–136 (2019)
17. Miller, R., Shanahan, M.: Some alternative formulations of the event calculus. vol. 2408, pp. 452–490 (2002)
18. Morgenstern, L.: Mid-sized axiomatizations of commonsense problems: A case study in egg cracking. Studia Logica **67**, 333–384 (2001)
19. Mueller, E.T.: Event Calculus Reasoning Through Satisfiability. Journal of Logic and Computation **14**(5), 703–730 (2004)
20. Okada, K., Mitsuharu Kojima, Satoru Tokutsu, Yuto Mori, Toshiaki Maki, Masayuki Inaba: Task guided attention control and visual verification in tea serving by the daily assistive humanoid hrp2jsk. In: 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 1551–1557 (2008)
21. Pednault, E.P.D.: Adl: Exploring the middle ground between strips and the situation calculus. In: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning. p. 324–332 (1989)
22. Reiter, R.: 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. pp. 359–380 (1991)
23. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 3rd edn. (2009)
24. Russell, S.J., Norvig, P.: Artificial Intelligence: a modern approach. Malaysia; Pearson Education Limited (2016)
25. Schiffer, S., Ferrein, A., Lakemeyer, G.: Reasoning with qualitative positional information for domestic domains in the situation calculus. Journal of Intelligent & Robotic Systems **66**, 273–300 (2012)
26. Shanahan, M.: A circumscriptive calculus of events. Artificial Intelligence **77**, 249–284 (1995)
27. Shanahan, M.: Robotics and the common sense informatic situation. In: European Conference on Artificial Intelligence (ECAI). vol. 12, pp. 684–688 (1996)
28. Shanahan, M.: An abductive event calculus planner. The Journal of Logic Programming **44**(1), 207–240 (2000)
29. Shanahan, M.: An attempt to formalise a non-trivial benchmark problem in common sense reasoning. Artificial Intelligence **153**(1), 141–165 (2004)
30. Williams, B.C., Ingham, M.D., Chung, S.H., Elliott, P.H.: Model-based programming of intelligent embedded systems and robotic space explorers. Proceedings of the IEEE **91**(1), 212–237 (2003)