Formal Methods in Robot End User Development: Progress, Gaps, and Opportunities

Suggested Citation: Yuna Hwang, Christine P. Lee, David Porfirio, Laura M. Hiatt and Bilge Mutlu (2026), "Formal Methods in Robot End User Development: Progress, Gaps, and Opportunities", : Vol. 13, No. 3, pp 1–18. DOI: 10.1108/FTROB-05-2025-0083.

Yuna Hwang, Christine P. Lee, Bilge Mutlu
Department of Computer Sciences,
University of Wisconsin-Madison
{yunahwang, cplee5, bilge}@cs.wisc.edu

David Porfirio
Department of Computer Science,
George Mason University
{dporfiri}@gmu.edu

Laura M. Hiatt
Navy Center for Applied Research in Artificial Intelligence,
U.S. Naval Research Laboratory
{laura.m.hiatt}.civ@us.navy.mil

now
the essence of knowledge
Boston — Delft

Contents

Formal Methods in Robot End User Development: Progress, Gaps, and Opportunities

Yuna Hwang[1], Christine P. Lee[1], David Porfirio[2], Laura M. Hiatt[3] and Bilge Mutlu[1]

[1]*Department of Computer Sciences, University of Wisconsin–Madison, {yunahwang, cplee5, bilge}@cs.wisc.edu*
[2]*Department of Computer Science, George Mason University, {dporfiri}@gmu.edu*
[3]*Navy Center for Applied Research in Artificial Intelligence, U.S. Naval Research Laboratory, {laura.m.hiatt}.civ@us.navy.mil*

ABSTRACT

End user development (EUD) is an emerging paradigm in human-robot interaction (HRI), involving end users both creating and maintaining robot programs themselves. Formal methods encompass a set of techniques that are useful to end user development, emphasizing the correctness of programs that are often created using natural, on-the-fly, and sometimes even sloppy end-user input. These techniques include, for example, correct-by-construction synthesis of robot programs given incomplete developer input, automated repair of these applications, and formal verification of these applications against sets of correctness criteria. Despite the increasing popularity of EUD and formal methods individually in robotics, there is no comprehensive review of how both have been successfully combined. Our primary

---

Yuna Hwang, Christine P. Lee, David Porfirio, Laura M. Hiatt and Bilge Mutlu (2026), "Formal Methods in
Robot End User Development: Progress, Gaps, and Opportunities", : Vol. 13, No. 3, pp 1–18. DOI: 10.1108/FTROB-05-2025-0083.

objective is thereby to review current practices for applying formal methods within robot EUD. To achieve this objective, we include adjacent areas of EUD research, such as natural language programming, and adjacent approaches to formal methods, such as automated planning. Recognizing that the integration of EUD and formal methods in robotics is still a nascent topic, our secondary objective is to identify opportunities for future research. Our review thereby provides a primer on how to conduct further integration of formal methods in robot EUD.

1

Introduction

Robots are becoming increasingly prevalent in daily lives. For example, *social robots*, namely robots that communicate social cues, are currently used within educational settings to assist teachers, parents, and children in classrooms and homes (*e.g.,* Rahman *et al.*, 2024; Cagiltay *et al.*, 2023). *Service robots* are found in many airports and hotels, greeting and assisting customers with simple tasks, and are also becoming widely accepted in elderly homes and care facilities (*e.g.,* Lee and Riek, 2023; Stegner and Mutlu, 2022; Kubota *et al.*, 2021). All of these robots assist humans by handling repetitive and time-consuming tasks, such as fetching objects, providing information, and offering support in educational, service, and caregiving settings. Although robot adoption promises significant benefits such as reduced human workload and increased efficiency, many challenges still exist. Current robot adoption requires professional robotics engineers to *program* (*i.e.,* transform instructions into precise, executable code) or *train* specific capabilities onto robot platforms, such as a pick and place behavior in a manufacturing setting. However, there is an increased demand for day-to-day robot end users themselves to tailor and adapt robot programs to meet their own specific needs. We can then ask: how can robot end users participate in the

robot programming process to make robots better suited to their own needs?

End user development (EUD) is a programming paradigm that supports the creation and maintenance of robot programs by end users, as opposed to programming experts (Vaiani and Paternò, 2024). Robot EUD *tools* refer to software applications that enable EUD, with the goal being to alleviate the technical barriers of programming for people without programming experience. Unlike traditional robot programming approaches that require a deep understanding of how robots operate, robot EUD tools abstract these details so that users can express desirable behavior of robots through intuitive means that align with their domain expertise.

Although EUD lowers the barrier to programming, challenges persist. Few EUD tools prevent their users from introducing mistakes into their programs, which can be costly once the programs are deployed in the physical domain. Additionally, robot capabilities are often not communicated directly to end users (Fisac *et al.*, 2020; Dragan and Srinivasa, 2014a; Dragan and Srinivasa, 2014b), which makes it challenging to identify what informed decisions a user is capable of making. Program maintenance and deployment are also difficult because end users are not always equipped with knowledge or assurance from the EUD tools that their program will execute correctly without errors, unexpected outcomes, or outright failures. End users may also wish to adapt programs created for other robot applications to their own domain, alleviating the need to create entire programs from scratch. However, few robot EUD tools explicitly support modifying existing programs.

A structured approach is needed to assist end users better navigate the complexities of creating *correct* robot programs. Fortunately, *formal methods* encompass a set of mathematical techniques that can help. In particular, *program verification* assists users by providing guarantees about their hand-crafted programs or highlighting where errors might occur during execution. Correct robot program creation can be supported through *program synthesis*, where given a set of end-user requirements, correct-by-construction (Kourie and Watson, 2012) programs are produced. *Program repair* can also be used to automatically fix errors in

existing programs, further supporting debugging and program adaptation. Formal methods extend beyond these three techniques, but in this paper, we focus specifically on *verification*, *synthesis*, and *repair* as being key to supporting the EUD activities of creating, analyzing, and maintaining robot programs.

In what follows, we ground our discussion of formal techniques that support robot programming within the EUD workflow. Program verification, synthesis, and repair are individually mapped to three EUD phases—program analysis, program creation, and program maintenance, respectively. Therefore, throughout this paper, when we refer to a particular formal method, we are also effectively referring to its corresponding EUD phase, and vice versa.

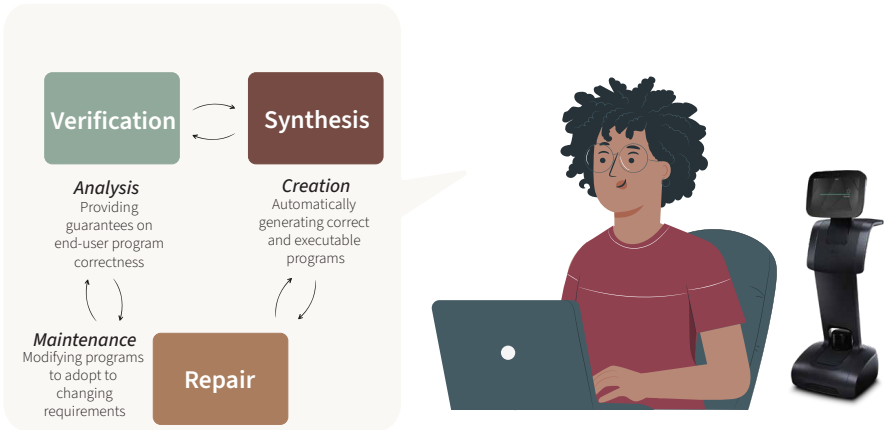

Figure 1.1: A programmer can leverage three formal methods techniques—verification, synthesis, and repair—while developing robot behaviors for platforms such as Temi[1]. Each formal methods technique is tied to an EUD phase, where verification supports analysis, synthesis supports creation, and repair supports maintenance. The EUD process does not require a fixed order; users may enter at different phases depending on their needs.

Program verification during the program analysis EUD phase provides end users with guarantees about the *correctness* of the programs they write. Verification addresses the challenges of end users being prone to inserting mistakes into their own programs and robot capabilities

[1] https://www.robotemi.com/

not being communicated directly to these users. EUD tools that apply formal verification techniques check all possible events in a program that might occur against a set of system requirements defined by the user (*i.e., specifications*) and robot constraints. Verification plays a crucial role when deploying robots in dynamic environments, where failures can be costly or even hazardous. By analyzing how a program will execute before deployment, existing EUD tools that employ verification are able to identify errors and provide confidence in program performance, thus ensuring reliability, all before the robot interacts with the real world.

Program synthesis during the program creation phase outputs robust and fully executable programs when given only partial specifications from end users. Similar to verification, the challenges synthesis addresses pertain to end users expressing program intent in imprecise or incomplete ways and introducing errors by applying robot functionalities in ways that do not align with their actual capabilities. Program synthesis generates correct-by-construction programs, such as those built upon refinement rules and logical formulae (Kourie and Watson, 2012), that are robust to these challenges while preserving the core end-user intent. Existing EUD tools that use synthesis techniques accept partial specifications from end users, such as input-output examples and *traces* (*i.e.,* sequence of human/robot events and tasks), in order to reduce the burden of specifying every scenario and corner-case that the robot could encounter.

Program repair during the program maintenance phase allows end users to modify robot behavior, either to fix execution failures or to adapt to changing requirements. The challenges repair addresses thereby pertain to program deployment and adaptation. Program repair modifies programs based on feedback received during robot deployment, which may occur if the robot is unable to manipulate an object with its end-effectors (*e.g.,* Kwon *et al.*, 2018), or if the robot notices that human operators are distracted during collaboration, to name a few examples. A rich source of feedback may be received when adapting existing programs to new use cases or to new end users. Existing EUD tools that utilize repair may accept end-user feedback directly or self-assess the causes of failure (*e.g.,* Han and Yanco, 2023; Norton *et al.*, 2022), and use this feedback to automatically generate program corrections.

Despite the proliferation of robot EUD tools and the application of formal methods in recent years, there has been no comprehensive review that focuses on the work done at the intersection of those two fields of research. In this paper, we thereby aim to review the current use of formal method techniques in robot EUD. Primarily, we characterize how each formal method technique is used across the lifespan of a robot program. Secondarily, we identify future research directions at the intersection of formal methods and robot EUD. To this end, we also examine adjacent areas with a long history, such as automated planning, and analyze how other techniques pursuing similar goals have been used in tandem with—or in place of—formal methods to enhance end-user experiences.

The order of content is as follows: In §2, we provide background on EUD as a development paradigm in human-robot interaction (HRI) and robotics. In §3, we describe the methodology used to select papers that fit the scope of this paper. We then motivate the application of formal method techniques to relevant EUD phases. §4 focuses on analyzing existing robot programs, where program verification is the most frequently adopted. §5 discusses program synthesis applied to program creation, and is further categorized into different approaches that emerge with the unique combinations of formal *representations* (*e.g., finite state machines)* and techniques (*e.g., SMT solving*). §6 presents program repair techniques applied to maintaining and adapting robot programs within the EUD phase. In §7 we provide further discussion points by addressing the gaps in current research, followed by §8 as a tutorial on how to apply formal method techniques to robot EUD as researchers. We conclude the paper in §9.

2

Background

The focus of our survey is on the advancement of end user development (EUD) systems in robotics and human-robot interaction (HRI) via the application of existing techniques in formal methods. Thus, with the advancement of EUD as our primary research focus, we provide a brief background of EUD in human-computer interaction (HCI), robotics, and HRI. We then provide a set of preliminaries for common formal method techniques and computational abstractions, or representations, necessary to realize these techniques.

2.1 End User Development

End user development (EUD) has roots in the wider field of enabling end users to program and customize human-computer interaction HCI systems. We thereby begin by describing EUD basics within the wider (HCI) research umbrella. Following that, we describe the state-of-the-art for EUD in robotics and HRI.

2.1.1 Overview of End User Development

Broadly, EUD entails the design of software artifacts by the end users of the artifacts themselves (Lieberman *et al.*, 2006). Whereas some work, particularly *end user programming* (EUP) concerns only the creation of these artifacts, EUD includes their entire lifespan—both creation and maintenance (Lieberman *et al.*, 2006). Within the entire lifespan of such artifacts, *end user software engineering* (EUSE) promotes the use of best practices in software engineering, such as testing and debugging, in the EUD pipeline (Barricelli *et al.*, 2019). The scope of this review paper includes all three foci—EUD, EUP, and EUSE—in robotics and HRI.

Historically, EUD has flourished within human-computer interaction (HCI). EUD has been applied to domains such as enabling end users to create, or program, graphical user interfaces (*e.g.,* Oney *et al.*, 2014), mobile applications (*e.g.,* Li *et al.*, 2017), networks of smart home devices and the Internet of Things (IoT) (*e.g.,* Fogli *et al.*, 2017), and recently, artificial intelligence (AI) agents (*e.g.,* Li *et al.*, 2018), to name a few examples. Within HCI, EUD continues to revolutionize how user interfaces capture user intent. Traditionally culminating in visual programming interfaces (Myers, 1986), EUD interfaces have more recently explored augmented reality (*e.g.,* Wang *et al.*, 2021; Chidambaram *et al.*, 2021; Wang *et al.*, 2020), natural language (*e.g.,* Li *et al.*, 2019), sketching (*e.g.,* Landay and Myers, 2001), and *programming-by-demonstration* (*e.g.,* Li *et al.*, 2017) as means of collecting user input. Within the aforementioned domains and methods of capturing user intent, much prior work has investigated different programming paradigms for EUD, referring to the process by which development occurs, how user input is represented, and how feedback is provided to end users. In recent years, *trigger-action programming* (TAP) has emerged as an EUD favorite, culminating in EUD tools such as "AutoTAP" (Zhang *et al.*, 2019) and "ImAtHome" (Fogli *et al.*, 2017). *Goal-oriented* paradigms represent another popular paradigm for IoT, in which device behaviors are expressed in terms of a desired end result (*i.e., goals*) rather than the steps necessary to achieve that end result (Noura *et al.*, 2018; Mayer *et al.*, 2016; Kovatsch *et al.*, 2015). An additional paradigm proposed by

Myers *et al.* (2004) is *natural programming*, an approach for designing programming tools and languages based on the natural ways the end users think about tasks.

2.1.2 Robot End User Development

EUD tools for robotics and HRI resemble their HCI counterparts in many ways, offering graphical (*e.g.,* Leonardi *et al.*, 2019; Alexandrova *et al.*, 2015), natural language (*e.g.,* Beschi *et al.*, 2019), and sketching (*e.g.,* Liu *et al.*, 2011) programming paradigms, to name a few examples. Due to the physical, embodied nature of robots, certain EUD paradigms have more emphasis in robotics, such as programming-by-demonstration (*e.g.,* Patton *et al.*, 2024; Huang and Cakmak, 2017), which often involve the user physically manipulating the robot (Akgun and Thomaz, 2016) or providing demonstrations via teleoperation interfaces (Hasan *et al.*, 2024). *In-situ* programming similarly has a greater emphasis in robotics, which overlays programming logic over the real world (Ikeda *et al.*, 2025; Senft *et al.*, 2021a), or a virtual depiction of the world (Huang *et al.*, 2020). In robot EUP, interacting with the robot at runtime to correct its execution is also often necessary (Porfirio *et al.*, 2020; Short *et al.*, 2019). Due to the multitude of uncertainties, significant risks, and high cost of deploying robots in the wild, formal approaches that ensure robot programs are correct *prior* to deployment, or capable of self-repair *during* deployment are especially important.

2.2 Formal Method Preliminaries

In this section, we present the preliminaries for program verification, synthesis, and repair. Before discussing each specific formal method, we give a formal definition of commonly used representations, such as finite state machines and logical formulae, to ground the reader's understanding in the relationship between raw user input, intermediate representations, and EUD output. We then outline the premises and problem definitions for program verification, synthesis, and repair, and discuss their objectives and the specific techniques used to achieve those objectives.

2.2.1 Representations

Formal methods use a myriad of different representations. Here, we describe two of these representations, which we view as frequently used in robot EUD. One of these is the **finite state machine** (FSM) and its family of closely related representations, which explicitly model the flow of how system state changes over time. We explain two variants of the FSM—the Mealy machine and the Moore machine. The other includes the family of **logical formulae**, which includes propositional logic, first-order logic, and variants of temporal logic. These representations play a central role in verification, synthesis, and repair by representing user specifications, intermediate representations used by the EUD tool, and the output of these formal techniques.

Finite State Machine (FSM) One representation used frequently within existing EUD tools is the **finite state machine** (FSM), which is a mathematical model consisting of *states*, which encompass the current state of the world, and *transitions*, which are temporal relations between states.

Informally, an FSM can be expressed as a *state diagram*, which is a visual representation of states and transitions (Sipser, 2013). Figure 2.1 is an FSM model of an on/off switch (Hopcroft *et al.*, 2001). Here, states are represented as circles and transitions as arrows between the states. Both transitions are labeled as "Push", where they denote external input that causes the state to change. The "off" state is denoted as the initial state, in which the system is first placed. Optionally, though not included in this example, are "accepting states", where no further input is accepted. In the context of robot EUD, an example state might be the robot's current observation or the current behavior that it is executing, and an example input that causes a change of state might be a user speaking to the robot.

Formally, an FSM can be expressed as the following (Sipser, 2013):

Definition 2.1. An **FSM** is a 5-tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$ where

1. Q is a finite set of **states**,
2. $q_0 \in Q$ is the **start state**,

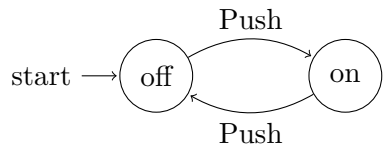

Figure 2.1: State diagram showing the transition between "off" and "on" states.

3. Σ is the **alphabet**, which is a finite set that consists of transition labels such as "Push",
4. δ: $Q \times \Sigma \to Q$ is the **transition function**,
5. $F \subseteq Q$ is the set of **accepting states**.

There are two main variants to an FSM (Hopcroft *et al.*, 2001): the *Mealy machine* and the *Moore machine*. Mealy machines produce output based on both the current state and the external input (*e.g.*, a human user's utterance), while Moore machines produce an output only based on the current state the FSM is in. For formal definitions of the Mealy and Moore machines, refer to Steffen *et al.* (2011).

Other variants of FSMs exist, for example, categorized by whether the FSM produces deterministic or non-deterministic outputs. Robot EUD tools also frequently use representations that closely resemble FSMs, such as transition systems, which can have an infinite number of states.

Logical Formulae End-user specifications on robot behaviors can also be expressed as constraints that the robot system must satisfy. Within verification, constraints are referenced as *properties*, because they describe the conditions that must hold true for the system to be correct. In synthesis and repair, constraints may guide the creation and modification of programs in a correct-by-construction fashion (Kourie and Watson, 2012), and help narrow down the large search space of satisfying programs.

Constraints can be expressed as logical formulae. In this portion of the article, we introduce the common types of logic used to express constraints and what those formulae look like. We first elaborate on the

fundamentals of propositional logic, and then expand our discussion to first-order logic (FOL) and linear temporal logic (LTL). Variants of LTL, such as metric temporal logic (MTL), are discussed to give the readers different flavors of logical formulae utilization in robot EUD. For a comprehensive review of logics, refer to Kress-Gazit *et al.* (2018).

Propositional logic (or Boolean logic) is a branch of logic that describes each state as a *proposition*, which evaluates to true or false, based on its meaning. For example, a state can be characterized as the status of the counting robot, expressed as $\pi_{counting}$ (Raman and Kress-Gazit, 2012). The proposition $\pi_{counting}$ can be evaluated to true if and only if the robot is counting numbers, and evaluated to false if and only if it is not. More complex truth evaluations can be made by combining atomic propositions with logical connectives such as $\neg$ (*not*), $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*if-then*), $\leftrightarrow$ (*if and only if*) (Ebbinghaus *et al.*, 1994). For example, $\pi_{counting} \wedge \pi_{bedroom}$ is true only when the robot is counting numbers and it is currently located in the bedroom (Raman and Kress-Gazit, 2012).

Constraints can also be written in **first-order logic** (FOL), or first-order predicate logic, which utilizes *functions* and *predicates* to denote the meaning or the relationship between entities. Because FOL is an extension to propositional logic, the truth evaluation of a constraint or a property extends to predicates (Tellex *et al.*, 2020). A first-order logic formula saying that the `agent` (*e.g.,* robot) and the `thing` (*i.e.,* object in question) are co-located can be expressed as—`location(agent) = location(thing)` (Schoen *et al.*, 2020).

Temporal logic formulae represent time-dependent constraints, which reason over the sequential ordering of states (*i.e., paths*).

Clarke *et al.* (2018) defines a path as the following:

Definition 2.2. A *path* π in M from a state s_0 is a finite or an infinite sequence of states $\pi = s_0 s_1 s_2 \cdots$ where for all $i \geq 0$, (s_i, s_{i+1}) is a transition in the model.

Within the path, each state is labeled with the truth value of the atomic propositions (Tellex *et al.*, 2020), and in Linear Temporal Logic (LTL, described below), the sequential ordering of states is enforced per

individual paths. Existing robot EUD tools leverage this characteristic of LTL to describe fairly complicated program paths.

Linear temporal logic (LTL) (Pnueli, 1977) is an extension of propositional logic, where it includes the propositional logic operators and a few other *temporal* operators that allow expressing assumptions about behaviors that change over time (Kress-Gazit *et al.*, 2018). These operators include `X, G, F,` and `U`, which mean "next", "always", "eventually", and "until", respectively.

Formally, the syntax and semantics of an LTL formula are as follows (Kress-Gazit *et al.*, 2018):

Definition 2.3. An **LTL formula** ϕ is recursively constructed from atomic propositions π according to the syntax:
$\pi := \phi \mid \neg\phi \mid \phi \vee \phi \mid \texttt{X}\ \phi \mid \texttt{G}\ \phi \mid \texttt{F}\ \phi \mid \phi\ \texttt{U}\ \phi$

An LTL formula has the following semantics:

1. ϕ is a proposition π.
2. $\neg\phi$ denotes it is not ϕ.
3. $\phi_1 \vee \phi_2$ denotes it either satisfies ϕ_1 or satisfies ϕ_2.
4. `X` ϕ denotes that ϕ is true in the **next** state.
5. `G` ϕ denotes that ϕ is true in **every** (*i.e.,* always) state in the path.
6. `F` ϕ denotes that ϕ is true in **some** (*i.e.,* eventually) state in the path.
7. ϕ_1 `U` ϕ_2 denotes that ϕ_1 is true **until** ϕ_2 becomes true within the path.

While LTL formulae only support the expression of time-dependent constraints abstractly, metric temporal logic formulae provide a more granular representation of time and the constraints surrounding it. **Metric temporal logic** (MTL) (Koymans, 1990) captures the "quantitative" timing properties that exist within real-time systems, and introduce the concept of "distances" (*i.e.,* time units) that characterize the timing of

events. For instance, MTL formulae can express constraints such as the traffic light turning red exactly 5 time units after the traffic light turns green (*i.e.,* $\texttt{G}(\texttt{green} \rightarrow (\neg \texttt{red}\ \texttt{U}_5\ \texttt{red}))$) (Ouaknine and Worrell, 2008). Formal syntax and semantics of MTL formulae are laid out in detail in Thati and Roşu (2005).

2.2.2 Program Verification

Being able to confidently assert the correctness of robotic systems is essential not only for expert software engineers, but also for end users, who may lack formal programming expertise. As more robotic systems are emerging in less-controlled, real world scenarios, the importance of safety and correctness guarantees has higher emphasis. Within formal methods, there are multiple approaches in which guarantees can be provided, program verification being one of them. Here, depending on the level of exhaustiveness, we situate program verification within a spectrum of techniques that achieve the same purpose of demonstrating correctness.

Verification is usually done at the analysis phase of robot programming. During or after the construction of a program, given (a set of) properties on desirable robot behavior, program verification exhaustively searches for violations of these properties in the program. The verifier can output a succinct "yes" or "no", though some verification techniques and tools (*e.g.,* theorem provers) provide more information, such as *counterexamples* (*i.e.,* example cases in which the scenario will fail), or formal proofs and axioms regarding program success and its associated properties.

Here, we explain a common verification technique, *model checking*, that performs exhaustive verification of states in program paths.

A model checker, employed within the model checking process, consists of three components (Clarke *et al.*, 2011),

1. a **model** of the program being verified, which in our case is a robot program. Deterministic programs are often modeled as transition systems.

2. formal **properties**, or the characteristics of the desired robot

program, written in temporal logic,

3. an automated **model checker**, an algorithm that exhaustively checks whether the model violates any of the properties.

Formally put, model checking can be described as (Clarke *et al.*, 2011):

Definition 2.4. Let M be a state-transition graph (*system model*) and let f be a temporal logic formula (program specification). The **model checking** problem is to find all the states $s \in S$ such that $M, s \models f$.

Given the finite state transition representation M and a temporal logic formula f, the model checking procedure can be summarized as discerning if all paths through the model satisfy f (*i.e.,* $M, s \models f$). If $M, s \not\models f$, some model checkers output counterexamples that witness the violation of f by M (Clarke *et al.*, 2018).

2.2.3 Program Synthesis and Repair

Program synthesis is an automated task that discovers programs that realize user intent on what the system must do. Gulwani (2010) introduces three key dimensions that capture program synthesis, which are, 1) expression of **user intent**, 2) **search space** of programs, and 3) **search techniques** that narrow down programs. Program repair similarly transforms existing, faulty programs into their fully correct counterparts. Both are similar in their aims and techniques; thus, we discuss both together.

User intent can be expressed in various modalities such as demonstrations, natural language, and input-output examples. In some cases, it can be directly known from the user intent how a specific input should be transformed into some output (*e.g.,* demonstrations of traces). On the contrary, there are cases where inference of the patterns or rules is necessary to capture the relationship between a set of unique inputs and outputs. Ambiguity is another challenge pertaining to the capturing and representation of user intent. For example, user intent expressed in natural language reduces the burden on users by eliminating the need to learn a new expression scheme. However, the drawbacks of

natural language include its high levels of ambiguity, posing challenges for translation into logical formulae (*i.e.,* the logics discussed in §2.2.1).

The **search space** of programs concerns the boundaries over which the desired programs will be searched. The search space in question can be significantly reduced by putting restrictions on the specific formats of the programs (*e.g.,* loop-free programs such as Mandelin *et al.* (2005)) on top of selecting programs that satisfy program specifications, *i.e.,* user intent expressed through system requirements. Smaller search spaces reduce the computational resources required to traverse each probable program in that space.

The **techniques** used for searching *candidate* programs in the search space span from brute-force, enumerative searching, machine learning-based searching, to the logical reasoning-based techniques (Gulwani, 2010), that operate on logical constraints. As described in the previous paragraph, enumeratively searching all probable programs is computationally taxing. Recently with the advance of machine learning algorithms, probabilistic inference techniques (*e.g.,* Li and Si, 2022) and neural network-based algorithms (*e.g.,* Kalyan *et al.*, 2018) are utilized to search through the program space effectively. Among the various search techniques, logical reasoning-based techniques are commonly used in robot EUD tools, often by integrating off-the-shelf Boolean satisfiability (SAT) or satisfiability modulo theories (SMT) solvers into these systems.

Boolean satisfiability (SAT) solvers are used in reasoning-based techniques. Within SAT solving, variables in the expressions can either have a value of 0 (`false`) or 1 (`true`). Formulae containing these variables are called propositional formulae (§2.2.1) and can be expressed in *Conjunctive Normal Form* (CNF), the standard representation required by most SAT solvers. Marques-Silva (2008) provides a definition of CNF:

> A CNF formula ϕ consists of a conjunction of clauses ω, each of which consists of a disjunction of literals. A literal is either a variable x_i or its complement $\neg x_i$.

The truth value, or the true *assignment* of the values is often denoted as μ (Audemard *et al.*, 2002). The SAT solver's goal is to find the truth assignment for each variable in a program. An example of a Boolean

formula in CNF is as follows: $\phi = (x \vee y) \wedge (\neg x \vee z)$, $\mu = \{x \mapsto true, y \mapsto false, z \mapsto false\}$ or $\mu = \{x \mapsto true, y \mapsto true, z \mapsto false\}$.

Satisfiability modulo theories (SMT) solvers extend SAT solvers to other existing theories, such as arithmetic theory (De Moura and Bjørner, 2011). As an example, an SMT solver would be able to handle variable assignments for statements such as $(t_1 \geq t2 + 3) \vee (t_2 \geq t_1 + 2)$.

3

Methodology

In this chapter, we provide details about the methodology used to select papers for this review. We employed a strict methodology to determine the set of papers we include in this survey. In particular, we utilized the Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA; Page et al., *2021) guidelines to ensure transparency and rigor in our selection process. In order to facilitate the systematic review process, we crafted a fixed search string that we then entered into several databases and querying engines, and then applied two sets of exclusion criteria to narrow down the number of papers subject to review. The literature search was conducted on May 29th, 2024, and after two iterations of applying exclusion criteria resulted in **32** papers that we consider for this article. Step-by-step process on literature screening is depicted in Figure 3.1.*

3.1 Paper Selection

To identify relevant papers, we used the search string:

("human-robot interaction" OR "robot") AND ("end user development" OR "end user programming" OR "graphical programming" OR "visual programming") AND "interface" AND ("formal method"

OR "formal analysis" OR "formal model" OR "model" OR "automata" OR "automaton" OR "domain-specific language" OR "synthesis" OR "verification" OR "monitor" OR "repair")

We justify these search terms as follows:

- *"human-robot interaction" OR "robot"*: We primarily aim to focus on papers that are written within the context of human-robot interaction (HRI) and robotics.

- *"end user development" OR "end user programming" OR "graphical programming" OR "visual programming"*: We aim to capture a range of tools that achieve interactive authoring and development of HRI and robotics scenarios. The terms *end user development* and *end user programming* are frequently used both in the HRI field and in the software engineering field (*e.g.,* Ko *et al.*, 2011). We add the terms *graphical programming* and *visual programming*, to collect for papers that describe robot programming interfaces in the computer science education (CSE) fields (*e.g.,* Chen and De Luca, 2016). We therefore include these terms to capture practical programming applications beyond end user development.

- *"interface"*: We aim to collect papers discussing interfaces that instantiate the EUD paradigm, rather than papers that introduce notional frameworks or pure algorithms.

- *"formal method" OR "formal analysis" OR "formal model" OR "model" OR "automata" OR "automaton" OR "domain-specific language" OR "synthesis" OR "verification" OR "monitor" OR "repair"*: We aim to collect papers that originate from or integrate formal method techniques, including model-driven formal method techniques or newly designed *domain-specific languages* (DSLs) within the software engineering realm. We also include keywords that talk about the specific methods within formal methods, such as *synthesis* and *verification*.

We entered this query into four database querying engines. The list of databases we utilized is: *Google Scholar*, *Scopus (Elsevier)*, *ACM Digital Library*, and *IEEE Xplore*. Search terms were found within

anywhere in the articles or their metadata, and we filtered out papers that were published before 2010. We exclude work that dates prior to 2010 due to observing a surge of EUD work in HRI beginning from 2010.

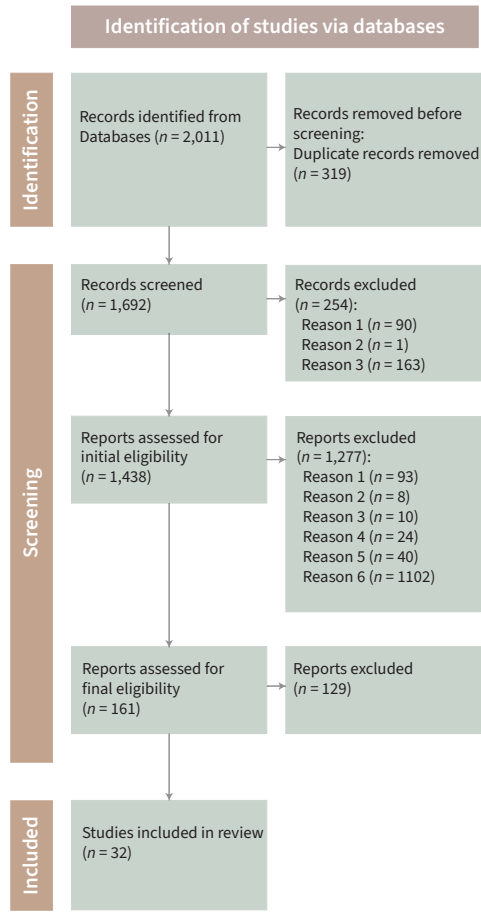

Figure 3.1: PRISMA flow diagram illustrating the selection process for studies included in this review article.

This resulted in 2,011 articles:

- Google Scholar yielded 8,150 articles, of which we considered the first 700 articles when sorted by relevance (a similar paper collection method was used in Ajaykumar *et al.* (2021)).
- Scopus yielded 779 articles.
- ACM Digital Library yielded 498 articles.
- IEEE Xplore yielded 34 articles.

Removing duplicates from the queried databases resulted in a total of 1,692 articles. From there, we performed screening of the collected articles. The exclusion criteria are as follows:

- Reason 1: Short papers that are less than 5 pages long ($n = 90$)
- Reason 2: Papers that are not written in English ($n = 1$)
- Reason 3: Indices for conferences and journals ($n = 163$)

After screening, 1,438 articles were subject to initial assessment of eligibility. During this phase, the full texts of the articles were reviewed. A set of exclusion criteria was applied to filter out papers that did not meet our scope. The exclusion criteria are as follows:

- Reason 1: Papers that make theoretical (*e.g.,* proposing notional frameworks, interaction design approaches) or empirical contributions (*e.g.,* evaluating an existing artifact) ($n = 93$)
- Reason 2: Survey papers that provide an overview of EUD, but not a specific EUD artifact or a formal methods technique ($n = 8$)
- Reason 3: Artifact papers but those built on top of existing programming frameworks such as LabVIEW ($n = 10$)
- Reason 4: Artifact papers but those that mainly consist of function calls for one-time use ($n = 24$)

- Reason 5: Masters/Ph.D. theses that include multiple artifacts ($n = 40$)
- Reason 6: Papers that are unrelated to robotics, or where robots are mentioned only peripherally and not as a central focus of the work ($n = 1102$)

After applying the above set of exclusion criteria, a total of 161 papers remained. We then conducted a final screening step, excluding papers that did not incorporate formal method techniques—specifically verification, synthesis, or repair—or that lacked any user-interface components. This resulted in a final selection of **32** papers that met all inclusion criteria. Figure 3.1 illustrates this literature screening process.

Even though many articles were retrieved in the initial stages, only a few subset of papers matched our selection criteria. This shows that the keywords appearing in query strings are often used informally or in ways that do not align with the formal methods concepts we target, leaving only a limited set of papers subject to review. Despite this small number, we conduct an in-depth analysis of these papers to provide a guideline-style account of the current research landscape.

Below is a full table of robot EUD studies that are included in this review article. To view the bibliography files for the surveyed studies, refer to the OSF link[2].

Table 3.1: *Overview of Robot EUD Studies. Papers listed in order of appearance.*

Paper	EUD Phase	Formal Method Type	Formal Method Goal
Schenkenfelder *et al.* (2023)	Analysis	Type checking	Ensuring Validity
Moser *et al.* (2014)	Analysis	Type checking	Ensuring Validity
Porfirio *et al.* (2018)	Analysis	Type checking, Verification	Ensuring Validity, Ensuring Correctness
Porfirio *et al.* (2023)	Analysis	Pre-/postcondition evaluation	Ensuring Validity
	Creation	Synthesis	Completing Program Structure

[2] https://osf.io/avqxt/overview?view_only=f2d64b88776c4e9e81e5549aa6d1cadb

Table 3.1 (continued)

Paper	EUD Phase	Formal Method Type	Formal Method Goal
Alexandrova *et al.* (2015)	Analysis	Pre-/postcondition evaluation	Ensuring Validity
Lucci *et al.* (2022)	Analysis	Verification	Ensuring Correctness
Askarpour *et al.* (2021)	Analysis	Pre-/postcondition evaluation, Verification	Ensuring Validity, Ensuring Correctness
Schoen *et al.* (2020)	Analysis	Pre-/postcondition evaluation, Verification	Ensuring Validity, Ensuring Correctness
Datta *et al.* (2016)	Analysis	Verification	Ensuring Correctness
Lyons *et al.* (2015)	Analysis	Verification	Ensuring Correctness
Lyons *et al.* (2012)	Analysis	Verification	Ensuring Correctness
Hurnaus and Prähofer (2010)	Analysis	Verification	Ensuring Correctness
Porfirio *et al.* (2019)	Creation	Synthesis	Completing Program Structure
Kubota *et al.* (2020)	Creation	Synthesis	Completing Program Structure, Imposing Program Constraints
Porfirio *et al.* (2024)	Creation	Synthesis	Completing Program Structure
Sauer and Henrich (2022)	Creation	Synthesis	Completing Program Structure
Porfirio *et al.* (2021)	Creation	Synthesis	Completing Program Structure
Porfirio *et al.* (2020)	Creation Maintenance	Synthesis Repair	Completing Program Structure Repairing for Adaptation
Liang *et al.* (2022)	Creation	Synthesis	Completing Program Structure
Liang *et al.* (2019)	Creation	Synthesis	Completing Program Structure
Gavran *et al.* (2020)	Creation	Synthesis	Imposing Program Constraints

Table 3.1 (continued)

Paper	EUD Phase	Formal Method Type	Formal Method Goal
Gavran *et al.* (2018)	Creation	Synthesis	Imposing Program Constraints
Chung and Cakmak (2020)	Creation	Synthesis	Parameterizing Programs
	Maintenance	Repair	Repairing for Adaptation
Racca *et al.* (2020)	Creation	Synthesis	Parameterizing Programs
	Maintenance	Repair	Repairing for Adaptation
Hagenow *et al.* (2024)	Maintenance	Repair	Repairing for Adaptation
Senft *et al.* (2021b)	Maintenance	Repair	Repairing for Adaptation
Meng and Kress-Gazit (2024)	Maintenance	Repair	Repairing for Generalizability
Holtz *et al.* (2020)	Maintenance	Repair	Repairing for Generalizability
Holtz *et al.* (2018)	Maintenance	Repair	Repairing for Generalizability
Mollard *et al.* (2015)	Maintenance	Repair	Repairing for Generalizability
Van Waveren *et al.* (2022)	Maintenance	Repair	Repairing for Generalizability
Jiang *et al.* (2017)	Maintenance	Repair	Repairing for Consistency

4

Review of Program Analysis within Robot EUD

In this chapter, we provide a review of robot EUD work that analyzes the ***correctness*** *of programs, using techniques from program verification. We take the perspective of examining which aspects of these programs are formally verified—for example, whether correctness pertains to the logical structure of programs, or the satisfaction of specified properties with respect to the programs.*

Within robot end user development (EUD), analyses of programs can be conducted through several different formal approaches, including **type checking**, **pre-/postcondition evaluation**, and **verification**. In robot EUD, variants of type checking and pre-/postcondition evaluation ensure the syntactic and semantic validity of robot programs. Verification is more complex, checking whether a program does what it is supposed to do functionally (*e.g.,* Kress-Gazit *et al.*, 2021; Wing, 1990). *Correctness* is represented as a set of pre-defined/user-defined properties that describe how a program should behave in a *functional* manner. For example, properties can encode how certain states must be reached eventually. These are called *liveness* properties (Lamport, 1977). Similarly, *safety* properties can be characterized as ensuring *nothing bad will happen*. Current EUD tools apply type checking, pre-/postcondition

evaluation, and verification to analyze end-user programs for validity and adherence to these properties, with the goal of identifying issues and prompting users to make the necessary modifications to ensure successful program creation and execution.

In what follows, we describe three common EUD approaches for how formal methods are applied to the analysis of robot EUD programs: (1) ***ensuring validity***; (2) ***ensuring invariant correctness***; and (3) ***ensuring temporal correctness***. Ensuring validity pertains to the use of "lightweight" formal methods (Kulik *et al.*, 2022; Jones *et al.*, 1996), such as type checking, to ensure that programs are syntactically valid. Ensuring correctness pertains to the use of verification techniques, such as model checking (Clarke *et al.*, 1986; Queille and Sifakis, 1982), to confirm that programs adhere to specified properties. We include common representations that are used for each approach, and exclusively focus on reviewing the *design-time* application of these techniques, as this is the most commonly observed pattern of application within robot EUD tools. Table 4.1 summarizes the types of formal method techniques for program analysis found in existing robot EUD literature.

Table 4.1: *Overview of Formal Method Techniques in Robot EUD.*

Formal Technique Type	**Representations**	**Formal Techniques in Practice**	**Paper Examples**
Ensuring Validity	Types	Type checking, pre-/postcondition evaluation	Schenkenfelder *et al.* (2023) Porfirio *et al.* (2023) Alexandrova *et al.* (2015)
Ensuring Correctness	Logical formulae (*e.g.,* first-order logic, linear temporal logic)	Model checking	Askarpour *et al.* (2021) Schoen *et al.* (2020) Porfirio *et al.* (2018) Datta *et al.* (2016) Lyons *et al.* (2015) Hurnaus and Prähofer (2010)

4.1 Ensuring Validity

Representations. In adapting type checking to robot EUD, a ***type*** might include labeled user-created robot behaviors, or labeled sensor signals that a robotic system could produce. Types are often used to enforce pre- and postconditions of robot behaviors, ensuring that behaviors are only connected in ways that make sense. In this way, types determine *which* operations are valid, and *how* they can be composed within a program, helping end users create correct and executable behavior flows.

Ensuring Validity in Practice. **Type checkers** used in Schenkenfelder *et al.* (2023) and Moser *et al.* (2014) prohibit users from connecting two welding-related functions in the graphical user interface when the robot output signal (*e.g.,* pulse dynamic correction) is incompatible with the welding machine's expected input (*e.g.,* possible range for pulse correction). A type checker in "RoVer" (Porfirio *et al.*, 2018) also ensures that the outputs of certain behaviors (*e.g.,* `Greeter`) are a subset of the allowable inputs for other behaviors (*e.g.,* `Ask`) before these behaviors are connected.

"Tabula" (Porfirio *et al.*, 2023) and "RoboFlow" (Alexandrova *et al.*, 2015) are other examples of EUD tools that check whether adjacent robot behaviors satisfy each other's **pre- and postconditions**. Tabula assembles user input to create a small, incomplete finite state machine (FSM; §2.2.1) where robot actions are represented as states. Because the user-provided input is incomplete, certain actions specified by the user might not have their preconditions satisfied. Tabula flags the unsatisfied preconditions and inserts necessary actions into the FSM to ensure satisfaction. For instance, if the user specifies an FSM with a "put" action, Tabula recognizes that "put" is preconditioned on the robot having grabbed something first, and accordingly inserts a "grab" action. RoboFlow similarly ensures the validity of robot programs by evaluating action pre-/postconditions. For example, before finalizing the "manipulation" action, RoboFlow checks whether physical landmarks that are referenced by this action are actually visible to the robot. In RoboFlow, postconditions output whether the manipulation procedure

has succeeded. Similarly, in Askarpour *et al.* (2021) and "Authr" (Schoen *et al.*, 2020), robot actions can only be added to the EUD editor when their pre-/postconditions align with those of adjacent behaviors.

4.2 Ensuring Correctness

The purpose of program verification is to ensure program correctness against a set of properties. The **correctness** of a program is defined as the "program doing what it is supposed to do" (Goldschlager and Lister, 1986). We refer to *invariant* correctness as ensuring that some property applies to every state of a robot program (known as invariants), in contrast to *temporal* properties that reason about the behavior of programs over time. Many existing robot EUD tools (*e.g.,* Lucci *et al.*, 2022; Askarpour *et al.*, 2021; Schoen *et al.*, 2020; Porfirio *et al.*, 2018; Datta *et al.*, 2016; Lyons *et al.*, 2015; Lyons *et al.*, 2012; Hurnaus and Prähofer, 2010) use temporal properties to verify correct ordering of robot actions within programs. Temporal properties often require more expressive logic than invariant properties.

In this section, we describe the different logics used to reason about invariant correctness and temporal correctness. We also categorize the types of properties (*e.g., liveness*, *safety*, and *fairness* properties) and discuss their significance in verifying correctness. Finally, we discuss how verification techniques are used in existing work. Although ensuring invariant correctness and temporal correctness sometimes relies on different kinds of logics, the underlying verification techniques (*e.g.,* model checking) can be applied in a similar manner, and we therefore discuss them together.

Representations Used to Ensure Invariant Correctness. EUD tools under this category use logical formulae to represent invariants. Examples include "VIPARS" (Lyons *et al.*, 2015; Lyons *et al.*, 2012), where the invariant is expressed in **first-order logic** (FOL; §2.2.1) and specifies that the robot must never go within one meter of an obstacle—that is, the system checks in every state whether the distance `q` satisfies $\texttt{q} > 1$. Hurnaus and Prähofer (2010) also use the following FOL formula—where $\neg$`(d.isDrilling()` $\wedge$ `d.rpm()<5000)` says that for every state,

drilling is only allowed when the driller speed is lower than 5000 rpm. Here, predicates such as `d.isDrilling()` express the current state of the robot. Because invariants are state-based properties, such predicates allow those invariants to be written and subsequently checked across all reachable states.

Representations Used to Ensure Temporal Correctness. Temporal properties evaluate program behavior across sequences of states. Predicate logic formulae are often used to represent preconditions of robot actions. Temporal logic formulae are also commonly used to express more complex logic as the operators capture rich patterns of behavior across sequences of states. Although there are many different types of temporal logic, here we focus on the common temporal logic types used in existing robot EUD—*linear temporal logic* (LTL; §2.2.1) and *metric temporal logic* (MTL; §2.2.1). LTL formulae specify whether action-occurrence and action-ordering properties hold within program traces, while MTL formulae extend LTL formulae by enforcing action-occurrence and action-ordering within specific time intervals.

Existing robot EUD tools use **predicate logic** formulae to verify satisfaction of pre-/postconditions of each robot action within programs. For example, "Authr" (Schoen *et al.*, 2020) verifies the precondition for the `grip` action—that the robot is not already gripping anything (¬`gripping()`)—before adding the `grip` action into the program. Lucci *et al.* (2022) also leverage predicate logic to express preconditions for a screwing operation in assembly settings, such as, `IsObjectInHand(Hand, Object)` ∧ `IsHandInVolume(Right, V2)`—ensuring that the screw action only happens when the `Object` is in `Hand` and `V2` is on the `Right`.

On the other hand, there are works that utilize temporal logic formulae to reason about more complex properties that span across multiple states. "RoVer" (Porfirio *et al.*, 2018) uses **linear temporal logic** (LTL) formulae to formalize social norms and subsequently uses these formulae to verify end user-authored interactions. The motivation lies in assisting end users to create robot behaviors that align with human expectations and prevent conversational breakdowns during interactions. RoVer builds on the idea that social norms, such as greeting others

or avoiding interruptions, closely reflect the appropriate timing and ordering of actions. For example, a greeting norm is enforced by verifying that the robot does not repeatedly issue greetings at the beginning of the interaction. Similarly, a speech norm can be enforced by prohibiting the robot from speaking at the same time as the user. In LTL, this would be expressed as $\texttt{G humanSpeaking} \longrightarrow \neg\texttt{robotSpeaking}$. Given LTL formulae for each social norm, RoVer verifies whether end-user programs adhere to this set of formulae, flags when violations occur, and provides feedback on what went wrong, such as highlighting actions that were added in the wrong order, or notifying the absence of a greeting in the beginning of the interaction.

Metric temporal logic (MTL) formulae in "Papyrus" (Askarpour *et al.*, 2021) encode properties that specify robot actions be executed within a certain number of time units. For instance, the MTL formula—$\texttt{dist}_{\texttt{rh}} < \texttt{dist}_{\texttt{thresh}} \longrightarrow \texttt{Futr(speed}_{\texttt{r}}\texttt{, 1)}$—enforces that the end-user program includes a time-bounded response, which is to stop the robot within one time step as soon as the human-robot distance is deemed unsafe.

Property Types in Ensuring Program Correctness Considering the large space of properties that can be expressed using a variety of different logics, these properties can be generally categorized based on the type of system behavior that they constrain—*liveness*, *safety*, and *fairness* properties.

"VIPARS" (Lyons *et al.*, 2015; Lyons *et al.*, 2012) verifies the **liveness** properties of end-user programs. In VIPARS, the success of a robot program is ensured by verifying that the robot reaches a certain goal waypoint and finishes every movement towards that waypoint in under 100 seconds. RoVer also verifies end-user programs against liveness properties. For example, the verifier ensures that the interaction eventually terminates, and ensures that the robot does not keep the human waiting forever without ever responding.

VIPARS also defines and utilizes **safety** properties, prohibiting the robot from approaching too close to an obstacle for both physical safety and adherence to mission success criteria.

Fairness properties are also often checked in robot EUD tools.

Verifying fairness means to check if "some event becomes possible infinitely often", while other events do not (Queille and Sifakis, 1983). RoVer addresses fairness by verifying that end-user programs do not include logic where the robot dominates the conversation by speaking twice or more times in a row. Similarly, Datta *et al.* (2016) verify that the expected robot actions occur only once and do not repeat.

Ensuring Correctness in Practice. Given a set of properties expressed in temporal logic, many EUD approaches apply a formal verification technique called **model checking** (§2.4). In model checking, every state within an FSM program is systematically inspected to confirm whether the properties specified in LTL, for example, are satisfied during program execution. The objective of model checking is to find a trace π within the transition system that *does not* satisfy a property, ϕ. The model checker outputs a confirmation if no states within all possible execution paths violate the property. In some cases, model checkers also provide counterexamples, namely execution paths that violate a set of properties. RoVer uses a PRISM model checker (Kwiatkowska *et al.*, 2011) to exhaustively examine every program trace and verify them against LTL formulae that encode social norms.

Existing model checking tools often use Satisfiability Modulo Theories (SMT; SMT solving in §2.2.3) solvers to find value assignments for each state in order to prove or disprove the correctness of all execution paths (De Moura and Bjørner, 2011). Solvers such as Z3 (De Moura and Bjørner, 2008) and a solver by Dutertre and De Moura (2006) are used in existing robot EUD tools, such as Schoen *et al.* (2020) and Hurnaus and Prähofer (2010). For solving MTL formulae, Zot solvers (Pradella, 2009) are often used.

One of the challenges during execution path enumeration is the *state explosion problem* (Clarke *et al.*, 2011), where the search space of the execution paths is so large that computation becomes intractable. Prior work addresses these issues, such as by propagating identical parameter values across states, observing that end users often chain states sharing the same parameter (*e.g.,* Schoen *et al.*, 2020). Papyrus also utilizes this approach to efficiently enumerate path segments of varying length.

5

Review of Program Creation within Robot EUD

In this chapter, we review robot EUD tools that create correct robot programs using techniques from program synthesis. As with formal techniques for analysis, the success of synthesis is contingent on its chosen representations—(1) the initial input, including user input and information that is missing from this input to be inferred by a synthesizer; (2) the intermediate representation that enables formal reasoning about the program being constructed; and (3) the program that is output by a synthesizer. This chapter describes robot program creation and synthesis through the lens of three different representation pipelines in robot EUD.

The program creation phase of robot EUD involves enabling users to develop robot behaviors that match their needs. In the context of formal methods, **program synthesis** focuses on automatically creating a robot program that is guaranteed to satisfy a user *specification*—a description of user intent expressed through system requirements. While program analysis checks whether an existing program meets a set of requirements, synthesis constructs such a program from the ground up. Often, synthesis leverages formal techniques for analysis, such as verification, to ensure that automatically constructed programs maintain correctness, or in other words, are *correct-by-construction* (Kourie and

Watson, 2012).

In this chapter, we focus on the nature of the initial input as the primary factor distinguishing various program synthesis work in HRI and robotics. We elaborate on the representation pipeline and the synthesis techniques used to transform this initial input into executable robot programs. We group the different synthesis approaches present in existing work into three major categories: (1) ***completing program structure***; (2) ***imposing program constraints***; and (3) ***parameterizing programs***. Completing program structure refers to transforming incomplete program structures—such as disjoint snippets specified by the end user—into fully formed, executable programs. Imposing program constraints refers to synthesis processes that begin from *constraints*, *i.e.,* end-user specifications that bound robot behavior. Lastly, parameterizing programs refers to synthesis approaches that begin from full program structures, which are then automatically parameterized by filling in missing variable values.

Table 5.1: *Overview of Program Synthesis Techniques in Robot EUD.*

Synthesis Type	Representations	Synthesis in Practice	Paper Examples
Completing Program Structure	Finite state machines (FSMs), logical formulae (*e.g.,* first-order logic, linear temporal logic formulae)	Automata learning, constraint solving (*e.g.,* SAT/SMT solving), CEGIS, automated planning	Sauer and Henrich (2022) Liang *et al.* (2022) Porfirio *et al.* (2019)
Imposing Program Constraints	Logical formulae (*e.g.,* linear temporal logic, description logic)	Controller synthesis, automated reasoning/planning (*e.g.,* PDDL planning)	Kubota *et al.* (2020) Gavran *et al.* (2018)
Parameterizing Programs	Finite state machines (FSMs)	Sketching, Bayesian filtering	Chung and Cakmak (2020) Racca *et al.* (2020)

5.1 Completing Program Structure

Creating correct programs begins with collecting and processing example behaviors of what the robot should do. Program synthesis in this pipeline involves composing incomplete program structures—such as end-user-provided, linear traces—and formalizing them into executable program structures, such as finite state machines (FSMs; §2.2.1). This approach

leverages sequence-like input representations to generate generalizable programs that capture and integrate the behavioral patterns across example inputs. Techniques such as automata learning and satisfiability solving (SAT/SMT solving in §2.2.3) are primarily used within this synthesis approach.

Representations. During the elicitation and gathering of end-user programming intent, end users may provide incomplete program structures, such as linear program **traces**. These traces serve as example execution paths through the not-yet-constructed full program, and are represented as input-output sequences, often of alternating states and events. States in a trace usually represent either a robot action/goal or its internal/environmental condition. For example, "Tabula" (Porfirio *et al.*, 2023) encodes robot actions, such as `grab(groceries)`, as states within individual traces that are provided to the system. External events or other information that the robot senses often denote the transition from state to state, such as human actions that trigger robot actions or responses. For example, "Synthé" (Porfirio *et al.*, 2019) models robot behavior traces as Mealy machines (§2.2.1), with states being internal conditions of the robot, while transitions are pairs of human actions and robot actions that modify those internal representations. "JESSIE" (Kubota *et al.*, 2020) is similar in that user input consists of sequences of robot actions, each of which may be conditioned on a specific stimulus occurring beforehand. With both systems, traces are very similar to, and can be easily operationalized as, FSMs without any branching or looping.

Logical formulae are often employed as intermediate representations in the synthesis pipeline to ensure that the synthesized program remains consistent with the user-provided traces. For instance, Synthé represents the desired program as a logical conjunction of separate user-specified traces, using **first-order logic** (FOL; §2.2.1) to encode demonstration traces and capture their shared common state transitions through logical conjunctions (the "and" operator, $\wedge$). With JESSIE, **linear temporal logic** (LTL; §2.2.1) properties encode sequences of robot actions and possible stimuli that these actions are conditioned on.

FSMs can also serve as incomplete program input, such as if the

transition between one state and another state represents a gap in program logic that must be filled with additional states and transitions. This is similar in concept to "Polaris" (Porfirio *et al.*, 2024), in which users provide a high-level FSM that encodes robot goals, such as `resident-alertedTo-lunchtime`, into FSM states. An automated task planner then splits each transition into multiple sub-states that must be met in order for the robot to proceed from one goal to the next. Note that while automated task planning is usually thought of as distinct from program synthesis, it behaves very similar to synthesis in the case of Polaris.

Completing Program Structure in Practice. Robot EUD approaches utilize a plethora of formal method techniques for synthesizing program structure from raw, possibly incomplete or partial user input. One such approach is *automaton minimization*, which yields compact FSMs from long traces of robot actions provided by the user. Rather than generating programs with emergent behaviors, the purpose of automaton minimization is to express lengthy user input more concisely. A related form of program synthesis observed in robot EUD is *automata learning*, which produces generalizable robot programs that handle cases beyond those explicitly captured by end-user input.

For incomplete input, some work uses *automated planning* as a synthesis technique. Examples include works in which an automated planner adds states to an FSM when end users skip key robot actions (thereby causing unmet action preconditions) or specify robot programs at such a high level that requires lower-level program details to be inferred (*e.g.,* specifying robot programs in terms of goals, from which the robot's actions must be inferred).

Some works define synthesis as a *constraint solving* problem, where they use logical formulae—such as FOL formulae—to formally define how user input must be assembled into the final program that satisfies as much of the user's intent as possible.

Because there is a large body of existing robot EUD works that fall under the category of *completing program structure*, this section organizes papers under distinct headers that reflect specific techniques used in each work.

Automaton Minimization Sauer and Henrich (2022) use an **automaton minimization** algorithm (Hopcroft *et al.*, 2001) to generate a compact FSM that represents a sequence of user-provided robot actions. In their work, end users physically manipulate the robot when demonstrating the action sequence. Every time a new haptic interaction is detected, a robot action is defined and is added as a state in the action trace. Next, program creation occurs by detecting overlaps in user-provided sequences in order to generate loop structures, thus "minimizing" the FSM.

Automata Learning While traces can be condensed to yield compact robot programs that faithfully represent user intent, some robot EUD tools utilize synthesis techniques that go one step further—inductively learning user input-output patterns from *multiple* traces to produce generalizable programs. "Figaro" (Porfirio *et al.*, 2021), for example, leverages **automata learning** (Steffen *et al.*, 2011) to search for FSM programs that accept multiple user-specified robot traces. These traces include the robot's trajectory in two-dimensional space and any discrete actions that the robot takes within these trajectories. In Figaro, traces directed toward the same destination are merged, regardless of the specific trajectories taken by the robot, allowing the program to generalize to new starting positions and unseen trajectories while still reaching the intended destination.

Synthé is another example work that uses automata learning. Synthé implements the *bodystorming* design method (Schleicher *et al.*, 2010) by prompting end-user design teams to role-play the back-and-forth interactions between a human user and a robot. Synthé models end-user demonstrations as traces in a Mealy machine (§2.2.1), in which human actions are FSM inputs and robot actions are FSM outputs. There are two transition functions in a Synthé FSM—an action transition function that defines how the robot reacts to human actions, and a state transition function that defines how the robot's internal state changes based on the human action. Synthé extends Neider (2014)'s automata learning technique and applies it to the Mealy machine domain. To learn iterative and cyclic robot behaviors present in end-user demonstrations, Synthé imposes an *inductive bias* towards minimal automata (this is an

additional constraint on synthesis, conceptually similar to automaton minimization).

While most existing applications of program synthesis in EUD treat robot action traces solely as examples of preferred behaviors from which to generate programs, some tools additionally collect negative examples and incorporate them into the synthesis process. For example, in Porfirio *et al.* (2020), negative examples, or counterexamples, are collected and used to exclude undesirable robot action traces from the final program. This approach is based on **counterexample-guided inductive synthesis** (CEGIS; Solar-Lezama *et al.*, 2006), where the synthesizer iteratively uses counterexamples to refine the program for continual learning and adaptation.

Automated Planning Automated planning is usually viewed as distinct from program synthesis; recently, however, some EUD tools have leveraged automated planning as a synthesis tool. Tabula utilizes program synthesis not only for assembling FSMs from individual user traces, but also for completing the individual traces themselves, where preconditions (*e.g.,* "grab") are automatically added before some actions (*e.g.,* "put") in the user-specified trace. Polaris is another example of an EUD tool that utilizes automated planning in a synthesis-like fashion. Strictly speaking, Polaris generates *plans* using an off-the-shelf Planning Domain Definition Language planner (PDDL planner; Fox and Long, 2003 and Hoffmann and Nebel, 2001), where the states in the goal automaton are used to generate complete, executable plans, *i.e.,* robot action sequences that achieve the goals set by the user's high-level hand-specified FSM. Through this design choice, Polaris alleviates user burden of exhaustively enumerating each robot action and instead allows users to simply express FSMs in terms of goals, such as `object is fetched` instead of individual actions—`move to object` and `grab-object`.

Constraint Solving Many EUD tools leverage constraint solving to complete a program's full structure from partial user input. In some cases, such as with Synthé, constraint solving is used in tandem with other approaches like automata learning. Specifically, as part of its automata learning pipeline, Synthé encodes end-user demonstration

traces into FOL formulae and performs MaxSMT solving (see SMT solving in §2.2.3) on them to generate compact programs. The goal of this is to maximize the number of demonstrations that jointly satisfy the inferred, FOL constraints, while leaving out traces that include possibly contradictory robot behaviors.

JESSIE constitutes another example of generating full program structures from incomplete user input via constraint solving. JESSIE involves a two-step synthesis process. First, the system synthesizes program constraints from user-provided action sequences, which are similar to traces (see §5.2 for a discussion on constraint synthesis). These constraints serve as an intermediate representation of user intent, and are used in the second step of the synthesis pipeline. From these constraints, the system utilizes an off-the-shelf synthesis tool named slugs (Ehlers and Raman, 2016) to generate an FSM robot controller that triggers relevant Robot Operating System (ROS) nodes (Wong and Kress-Gazit, 2017). JESSIE thereby uses constraint solving to translate incomplete user input into fully executable and correct-by-construction robot programs.

Instead of encoding intermediate logical formulae that capture program-wide logic, other robot EUD tools extract constraints bound to each robot action-level (*e.g.,* pre-/postconditions), and from these, create generalizable programs. For instance, "iRoPro" (Liang *et al.*, 2022; Liang *et al.*, 2019) generates pre-/postconditions expressed in FOL from robot action sequences, which describe how the robot's world (*e.g.,* object positions) has changed between robot actions, expressed such as `obj is on A`. These inferred intermediate representations are then passed to an off-the-shelf PDDL planner, which generates robot action plans that remain robust in new environments, as long as the pre- and postcondition formulae are satisfied.

5.2 Imposing Program Constraints

While some synthesis pipelines accept sequential input, like end-user demonstration traces, others take a more declarative approach, accepting user-defined program *constraints*. These constraints guide the creation of robot programs by setting boundaries without explicitly defining

program structure or flow. They limit robot behavior by, for example, specifying a partial order over actions rather than prescribing a full action sequence. Robot EUD tools that fit under this category often utilize temporal logic formulae to describe the required sequencing and timing of robot actions. These formulae may be the product of synthesis itself (*e.g.,* translating a user's natural language description of robot behavior to temporal logic) or might be used as input to a program synthesizer or an automated planner for further program generation, consequently yielding automata-based artifacts or step-by-step robot action plans.

Representations. Constraints are often expressed via **logical formulae**. These logical formulae tend to be intermediate representations within the synthesis pipeline, usually following specific branches of logic. In particular, linear temporal logic (LTL; §2.2.1) is frequently used to encode time-dependent constraints. As an other example, description logic (DL) is used for describing ontologies within specific domains (Baader *et al.*, 2008). Combined with principles from set theory, DL allows for the representation of relational constraints that govern robot programs.

Imposing Program Constraints in Practice. Synthesizing program constraints involves translating raw user input into logical formulae. The logical formulae may then be used for additional synthesis, or to guide an automated planner in selecting the appropriate actions for a robot to perform.

JESSIE is an example of an EUD tool that imposes program constraints. The system involves a two-step synthesis process—translates raw user input from a tangible authoring interface into **LTL formulae**, and subsequently generating a finite state machine (FSM; §2.2.1) robot controller from the LTL. We have already discussed this second step in §5.1. This section now focuses on the first step.

The system's tangible interface consists of a set of cards, where each card either represents a robot-initiated activity or sensing events that precede robot reactions, such as greeting people with mild cognitive

impairments—PwMCI—when they touch the robot. Here, the clinicians treating the PwMCIs place the activity cards in a specific order, thereby expressing the temporal ordering of the activities. The clinicians can express other types of constraints, too, such as by guiding the robot to display a congratulatory notification to the PwMCI only when they completed certain activity modules. These constraints are then translated into LTL formulae, which are then used for controller synthesis. In this way, JESSIE employs LTL as an intermediate synthesis representation.

As another example of imposing program constraints in practice, "LTLTalk" (Gavran *et al.*, 2020) uses a multi-objective optimization problem modulo SMT solver (Bjørner *et al.*, 2015) to search for candidate LTL formulae that best capture user intent. The input of LTLTalk are user demonstration traces and natural language descriptions that contextualize the demonstrations. For example, a user may instruct a robot to take one red item from the coordinates (7,4) by demonstrating the action and providing a corresponding description. Given these inputs, the system first generates extra samples from the user-given trace and then constructs a set of candidate LTL specifications through an SMT-based search (see SMT in §2.2.3) over all bounded-size formulae consistent with those samples. During this search, the solver gives highest priority to the objective that measures how well each candidate LTL formula aligns with the natural language description. The user, after reviewing the possible LTL formulae, is prompted to choose a single LTL formula that best fits their intentions. In order to facilitate this selection process, robot behavior traces that instantiate different LTL formulae are presented to the end user so that they can decide upon an LTL formula that best fits their intent. This approach follows the **distinguishing examples** approach (Gulwani, 2012; Jha *et al.*, 2010) in formal methods, where different inputs are used to determine whether candidate programs behave differently, allowing the program synthesizer to rule out incorrect specifications. After the final LTL formula is chosen, LTLTalk then invokes an off-the-shelf planner to generate step-by-step robot actions that satisfy the LTL constraint.

Lastly, "TOOL" (Gavran *et al.*, 2018) uses automated AI reasoners and planners to generate robot actions given constraints that are formulated in logical formulae, namely **description logic** formulae (DL;

Baader, 2003). TOOL allows end users to express these constraints through a domain-specific language (DSL) designed for describing relationships between elements in robot manufacturing tasks. Users define how the assembly items, workers, skills, resources, and tools in a manufacturing task are related to each other rather than describing how each element is used within an action sequence. A user's TOOL DSL code is then translated to a set of DL formulae that capture the inclusion relationship between the elements. For example, skills might consist of inclusion relationships in DL such as `gluing` $\subseteq$ `bonding`. Inclusion statements such as `skillOf_robot1` $\times$ `robot1` $\subseteq$ `canBeHandledByWorker` can model worker capabilities. Once all the relational constraints are defined and translated into corresponding DL formulae, step-by-step plans are generated using two off-the-shelf tools. First, an automated reasoner (Matentzoglu *et al.*, 2017) checks and matches resources to the skills. Then, an off-the-shelf PDDL planner is used to generate action sequences for both human and robot workers, factoring in all constraints defined by the user.

5.3 Parameterizing Programs

Some existing robot EUD tools adopt a parameterization-based representation pipeline, in which the synthesis process begins from full, template-like program structures, and lower-level program details are subsequently modified. The template-like program structures used within this approach accommodate both uncertainty and user interactivity and support incremental improvement over time, while at the same time preserving the performance and fluency of the of overarching program structure at runtime.

Representations. Full program structures are often represented as finite state machines (FSMs; §2.2.1) but with **holes** in place of low-level details, such as the *parameters* that guide robot execution (*e.g.,* the velocity of the robot in a `move` action). Similar to how FSMs are used in §5.1, we can think of FSM states as representing robot actions, while transitions encode the conditions that trigger these actions. In extending this representation, parameters set the thresholds or continuous values

that define those conditions. For instance, "SoRTSketch" (Chung and Cakmak, 2020) uses a Mealy machine representation of robot programs (§2.2.1). Within this representation, SoRTSketch determines the robot's reaction to user disengagement via a transition function that encodes the relationship between the user's gaze direction and the maximum angle of engagement. Depending on the disengagement parameter, the program will trigger either a "wait" action or a continuation of the interaction.

Parameterizing Programs in Practice. EUD tools that adhere to this synthesis pipeline use machine learning-based approaches from multiple iterations of explicit/implicit feedback from the users to infer the distribution or the exact values of program parameters. Although these approaches do not strictly fall under formal methods, we include them in this review article because the boundary between learning-based techniques and inductive program synthesis is often blurred, and these approaches still realize concepts closely aligned with formal methods.

SoRTSketch applies an approach that is similar in concept to **sketching** (Gulwani, 2012; Solar-Lezama, 2008) in formal methods. In formal methods, sketching utilizes SAT-based inductive synthesis techniques (see SAT solving in §2.2.3) to fill in the values of program (the *sketch*) holes. While adhering to the core idea of sketching in terms of filling in holes, SoRTSketch uses a different approach to SAT-based inductive synthesis—Bayesian Filtering to infer program modifications from *implicit* user feedback (Chen *et al.*, 2003). Whenever end users provide implicit feedback that indicates that a robot action transition is incorrect, the transition parameters are subsequently modified. For example, such feedback might be operationalized through "Next" and "Go back" buttons. The Next button implies that the robot missed a transition and should proceed to the next action rather than remain in the current state. The Go back button may be issued when the user thinks that the robot should stay longer in the previous state before transitioning to the current one. Both Next and Go back feedback ultimately pertain to the threshold values defined for the current transition function. This feedback adjusts the robot's transition behavior by refining these parameters, thereby ensuring that the robot's actions align with user

expectations.

Racca *et al.* (2020), on the other hand, accept more explicit feedback from the end user, such as "higher", "lower", and "fine", for tuning low-level robot action parameters. These parameters might include the end-effector's translational speed or the grasping force during a pick-and-place task. Similar to SoRTSketch, Racca *et al.* (2020) use a Bayesian estimation approach, which is combined with *expected maximum divergence* for parameter estimation (Roy and McCallum, 2001). Although this work primarily follows machine learning techniques, specifically the *active learning* paradigm (Settles, 2009), rather than formal methods, its use of "informative queries" to elicit significantly different parameter ranges bears similarity with the notion of **distinguishing input** (Gulwani, 2012) and the formal method techniques surrounding it. In formal methods, a *distinguishing input* refers to a generated input that yields different outputs. Racca *et al.* (2020) compute the informativeness of each query and presents queries to the user. Once users settle upon feasible parameter ranges through multiple query iterations and the parameter values eventually converge, the resulting program consisting of robot action sequences is yielded with final parameter values.

6

Review of Program Maintenance within Robot EUD

In this chapter, we provide a review of EUD work that addresses maintaining existing robot programs using techniques from program repair. We ground our discussion based on the program repair activities suggested by Le Goues et al. *(2019)—(1) fault localization, which identifies sources of errors or violations through analysis of execution traces (Lou* et al.*, 2020), (2) patch generation, which produces potential fixes, and (3) patch validation, which ensures the correctness and effectiveness of the proposed patches (Zhang* et al.*, 2023). This chapter analyzes robot EUD tools that engage in these activities for program maintenance and repair.*

While program verification and synthesis are useful tools for the creation and analysis phases of robot end user development (EUD), they do not account for the evolving correctness requirements encountered during program maintenance. The need for modifying existing robot programs arises when robots are deployed in the real world and encounter situations that prevent them from adhering to their original requirements, experience technical limitations, or receive incompatible requests from multiple users (*e.g.,* Das *et al.*, 2021). Correctness criteria may also change when users place robots in novel scenarios, or in the

case when user preferences evolve over time. Addressing these challenges necessitates continual program maintenance.

In this chapter, we categorize approaches that EUD tools use for program repair into (1) ***repairing for adaptation***, (2) ***repairing for generalizability***, and (3) ***repairing for consistency***. Our review is primarily based on the first two categories, particularly on handling evolving application contexts (*i.e.,* generalizability) and personal preferences (*i.e.,* adaptation). Not many EUD tools fit in the consistent-repair-generation category, but we still include them in our review to provide readers with the full scope of works present in current robot EUD literature.

For each repair objective, we first describe how faulty programs, their underlying *faults*, and the corresponding *patches* are represented. We then outline techniques that operate on these representations to facilitate the three repair steps described above. For an overview of the types of program repair and their corresponding examples, refer to Table 6.1.

Table 6.1: *Overview of Program Repair Techniques in Robot EUD.*

Repair Type	Representations	Repair in Practice	Paper Examples
Repairing for Adaptation	Counterexamples, holes in a program sketch	Fault localization, sketching, CEGIS, model checking	Chung and Cakmak (2020) Porfirio *et al.* (2020)
Repairing for Generalizability	Logical formulae (*e.g.,* first-order logic, linear temporal logic)	SMT solving	Meng and Kress-Gazit (2024) Holtz *et al.* (2020) Holtz *et al.* (2018)
Repairing for Consistency	Invariants	Application of transformation rules	Jiang *et al.* (2017)

6.1 Repairing for Adaptation

Repairing programs for adaptation begins with mechanisms for expressing faults in existing robot programs, such as expressing uncertainty and low-level variability within program structures. Considerations for future adaptation often arise from the evolving nature of robot tasks or from user-specific preferences. In existing robot EUD tools, finite state

machines (FSMs; §2.2.1) appear to be the primary representation of the programs subject to adaptation, while fault representations (*e.g.,* counterexample traces) and patch representations (*e.g.,* holes in a program sketch) capture the necessary modifications to be made and guide which techniques—such as counterexample-guided inductive synthesis (CEGIS; Solar-Lezama *et al.*, 2006) and constraint solving (see SAT/SMT solving in §2.2.3)—can be used to implement these modifications.

Representations. When repairing for adaptation, the programs that get adapted typically assume an **FSM** representation. In Porfirio *et al.* (2020), FSM programs are adapted to better fit the surrounding environment and social context. The interactions between users and the robot that lead to these modifications are represented as FSM traces. Traces that receive poor user ratings denote **counterexamples** to be removed via program program repair, whereas traces that receive good user ratings represent the positive examples that should not be removed. "SoRTSketch" (Chung and Cakmak, 2020) also utilizes FSMs as default program representation. Here, candidate fixes are represented as **holes** within the program, which are later filled with parameter values derived from end-user preferences in social interactions (*e.g.,* how long the robot should wait upon user disengagement). These parameters are refined through repeated feedback, indicating how they can be further adjusted.

Repairing for Adaptation in Practice. Robot EUD tools for repair, specifically for behavior adaptation utilize fault localization and sketching techniques that aim to identify and repair specific elements of the programs. In formal methods, fault localization schemes rely on heuristics to explore the repair space. In Porfirio *et al.* (2020), fault localization prioritizes which fixes to apply by ranking FSM transitions and states that appear the most in low-scoring interaction traces, *i.e.,* counterexamples. The localization of potential fixes is done in addition to fault localization, in which the repair process also prioritizes the retention of transitions and states that occur the most in high-scoring programs. These two optimization techniques operate within the **CEGIS** loop, through which robot programs applicable to new contexts are derived. Complementing this process, **model checking** (§2.4) is used as a patch

validation step, where it ensures that the repaired program does not violate any context-free constraints, such as social norms in this case.

SoRTSketch uses a similar approach to **sketching** to support adaptive program repair for personalization and contextual fit. Similar to fault localization approaches, only a small subset of program components—those most relevant to user interaction—are subject to modification, *i.e.,* patch generation. In this approach, candidate fixes are represented as holes in a program, specifically transition functions that are partially defined. The holes in these partially defined transitions consist of undefined thresholds for activating FSM state transitions. Contrary to using a SAT solver (see SAT solving in §2.2.3) to derive the values of the holes as in the original sketching method, SoRTSketch utilizes Bayesian inference to iteratively estimate parameters that better align with user preferences and situational context. Through this patch generation process, the system refines behaviors such as how the robot responds to user disengagement during a guided activity, yielding personalized yet stable social interactions.

Beyond the aforementioned works that adopt repair techniques grounded in formal methods, many existing robot EUD tools incorporate human-in-the-loop repair mechanisms for program adaptation. These works contribute algorithms and interaction techniques particularly for timely fault localization and patch generation. For instance, Racca *et al.*, 2020 propose an algorithm for efficient fault localization. Inspired by *active learning* techniques in machine learning (*e.g.,* Settles, 2009), the system employs informative queries to narrow the acceptable range of parameter values for different contexts—for example, applying different forces depending on object contact. Interaction techniques such as experts providing real-time fault corrections in high task-variability settings (Hagenow *et al.*, 2024) and modifying task parameters on-the-fly (Senft *et al.*, 2021b), further demonstrate how adaptive program repair can occur in real-time settings under human guidance.

6.2 Repairing for Generalizability

EUD tools that seek generalizability of logic during program repair use intermediate representations expressed as specific logical formulae to

apply reasoning to unseen environments, situations, and robot tasks. Once such intermediate representations are specified, methods such as constraint solving enable both the automatic generation and validation of generalized repair patches, ensuring correct-by-construction (Kourie and Watson, 2012) synthesis and robustness across varying contexts.

Representations. Robot program repair for generalizability operates over **FSMs** (§2.2.1) or other specifications that define the desired robot behaviors given environmental assumptions. For instance, Meng and Kress-Gazit (2024) modify high-level specifications expressed in **logical formulae**—such as linear temporal logic (LTL; §2.3)—that formally describe the relationship between robot tasks, skills, and environment assumptions. The representation used for the violation expression is a triplet—the set of environment propositions that are true before the robot acts, the robot skills being executed, and the change in the environment after skill execution. The program fix is symbolically represented, including relaxed environment assumptions and new robot skills that are synthesized from existing ones. Using a similar approach, repair in Holtz *et al.* (2020) and Holtz *et al.* (2018) occurs at the FSM level, where faulty conditions are directly specified by end users who highlight incorrect states within the FSM. The fix for repairing such programs involves intermediate specifications, such as FSM transition functions expressed in first-order logic (FOL; §2.2.1).

Repairing for Generalizability in Practice. Several robot EUD approaches transform program repair into a **constraint solving problem** or a learning problem. For example, Holtz *et al.* (2020) and Holtz *et al.* (2018) employ a Satisfiability Modulo Theories (SMT) solver (see SMT solving in §2.2.3) to extract logical formulae expressed in FOL and produce programs with updated parameter values that generalize better across contexts. Meng and Kress-Gazit (2024) do not directly use constraint solvers to yield repaired programs; however, they leverage them to determine whether repair is needed. The SMT solver diagnoses whether an LTL specification—which encodes assumptions about the environment and the robot task—needs to be rewritten. If an assumption relaxation is necessary, the robot specification is rewritten with

a logical disjunction, further including the newly written environment assumptions. In addition to modifying specifications, new robot skills are created by modifying pre-/postconditions of existing skills.

Existing robot EUD tools also utilize learning techniques to acquire generalizability in program logic. Although these do not strictly adhere to formal methods, we introduce these works briefly to demonstrate program repair applications within the current HRI landscape. In Mollard *et al.* (2015), constraints are generated through encoding key events of end-user demonstrations using a conditional random field model (CRF; Baisero *et al.*, 2015) in machine learning. Using these constraints, end users can further specify relationships between new objects in the robot workspace. In Van Waveren *et al.*, 2022, robot programs are iteratively generated through reinforcement learning (RL) techniques. In order to prevent repeated failures in the course of learning, however, *shields* (Alshiekh *et al.*, 2018) that are applicable to many failure types are generated from non-expert feedback, which includes action refinements, alternative action suggestions, and forbidding of specific actions. These shields are then used to repair existing programs which failed previously.

6.3 Repairing for Consistency

Robot program repairs often pursue structural and syntactic consistency to which the fix is applied. In this repair objective, patches are specified and generated based on pre-defined categorization of fault types, such as those that pertain to program *invariants*. *Transformation rules* that correspond to these fault categories are then applied to systematically modify the affected program components while preserving overall structure and intent.

Representations. Robot EUD tools that seek repair consistency often depend on **invariants**, or "statements that are true at specific points of program execution" (Nelson, 1980), to discern which specific fixes to apply in a consistent manner. Jiang *et al.* (2017), for example, infer and synthesize system invariants observed from Robot Operation System (ROS) message-passing traces and uses these invariants—such as data flow within traces, fixed frequencies of ROS messages—to determine the

type of pre-defined recovery actions when invariant violation has been observed. While invariants are also used within generalizable program repair approaches (*e.g., invariant generation*), in this repair approach, we highlight the role of invariants as anchors for consistent repair schemes—allowing pre-defined fix patterns to be applied efficiently once a violated condition is identified.

Repairing for Consistency in Practice. Formal method techniques in this type of program repair often use **transformation rules** (*e.g.,* Kim *et al.*, 2013) to yield consistent repair behaviors. Although this mechanism is not fully instantiated in Jiang *et al.* (2017), the work instead focuses on the inference of invariants from ROS traces. This work lists out a few fixed sets of recovery actions (*e.g.,* unregister an unknown publisher if the "architecture" invariant is violated) based on the specific violated invariant type.

7

Discussion

In this chapter, we discuss the promises, gaps, and opportunities for the integration of formal methods into robot end user development (EUD), drawing on insights from our review. First, we summarize the promise of formal methods for robot EUD. Following descriptions of what each technique could achieve across robot EUD phases, gaps in current research are highlighted, offering directions for future work. We conclude this chapter by outlining long-term directions, such as integrating human-centered design approaches and applying lightweight formal methods to robot EUD.

7.1 Promise of Formal Methods for Robot EUD

The application of formal method techniques has great potential for a wide range of problems in robot EUD. Program verification, synthesis, and repair techniques can support automated analysis of robot programs, create generalizable, correct-by-construction programs (Kourie and Watson, 2012), and help maintain and adapt programs in the long run.

7.1.1 Promise of Verification for Program Analysis

First, the program-analysis phase of the EUD life cycle can benefit from **verification** techniques, which checks whether a program does what it is expected to do. This process thus generates useful insights into the guarantees of end-user-created programs. By leveraging verification within EUD tools, end users can avoid time-consuming processes of manually debugging errors that impact system performance. In social robotics, formalizing commonly adopted social norms (*e.g.,* Porfirio *et al.*, 2020; Porfirio *et al.*, 2018)—often overlooked when end users focus on program functionality—and verifying their compliance in user-created programs through model checking can reduce interaction breakdowns during real world execution. Guarantees on system performance (*e.g.,* Lyons *et al.*, 2012; Hurnaus and Prähofer, 2010) could also be useful in industrial and collaborative robotics, where verification can model and monitor specification violations on safety-critical systems. For instance, the sequencing and timing of actions can be precisely defined, ensuring that the robot's contributions are well-coordinated with and seamlessly integrated into existing workflows.

7.1.2 Promise of Synthesis for Program Creation

The program-creation phase of the EUD cycle can also greatly benefit from **synthesis** techniques that generate programs from end-user specifications. Regardless of the types of end-user specifications, synthesis can be used to create generalizable, correct-by-construction programs through automated satisfaction checks (Kourie and Watson, 2012), which yields executable robot programs that operate safely in the real world. Program synthesis approaches support program creation by capturing user input, transforming it into necessary intermediate representations, and applying techniques, such as constraint solving (§5.2) and automata learning (§5.1), to generate robot programs that meet end-user specifications.

Program synthesis enables the processing of diverse forms of end-user input—ranging from sequential data such as robot actions (*e.g.,* Liang *et al.*, 2022; Sauer and Henrich, 2022) to high-level declarative constraints (*e.g.,* Kubota *et al.*, 2020; Gavran *et al.*, 2018)—by translating them into

intermediate representations that can be mathematically analyzed and used for correct-by-construction program generation. This flexibility in accepting different input modalities is particularly important in human-robot interaction (HRI) and robotics, as the elicitation and processing of end-user intent are key elements of creating useful EUD tools. Automatically creating generalizable programs via methods such as automata learning and constraint solving can help produce programs that match end-user intent while abstracting away implementation details. The correctness-guaranteeing optimization mechanisms in these techniques can contribute to creating compact and reliable programs across different robot applications.

7.1.3 Promise of Repair for Program Maintenance

The robot program-maintenance phase of the EUD cycle—fixing and adapting initial programs via **repair**—also holds significant potential. End users may seek to modify robot programs to meet their evolving preferences, address previously unconsidered scenarios, or overcome technical limitations of different robot platforms. Similar to synthesis, repair techniques generate programs that are correct with regard to user specifications, only with the new goal of retaining the original overarching program intent while satisfying new or updated user specifications.

Because repair techniques directly interface with robot program failures that arise across diverse scenarios and user preferences, they can be directly integrated into HRI and robotics contexts through EUD tools that keep end users in the loop. End users can rate their satisfaction with the interactions executed through the program (*e.g.,* Chung and Cakmak, 2020; Porfirio *et al.*, 2020) or pinpoint particular elements, such as robot actions, within the programs they find undesirable (*e.g.,* Holtz *et al.*, 2020; Holtz *et al.*, 2018). These feedback mechanisms ensure that programs built with EUD tools remain grounded in firsthand end-user experiences with the robot. Over time, such repair techniques hold promise to improve the contextual fit of robot programs, and can be used to enhance coordination and alignment in both social and industrial scenarios while reducing maintenance costs through quick, *in*

situ correction of faulty program elements.

7.2 Research Gap in Applying Formal Methods to Robot EUD

Despite the great potential formal methods hold for robot EUD, several research and resource gaps limit the impact of these approaches. Specifically, our review has identified gaps in (1) ***the lack of libraries, implementations, and best practices*** to facilitate the adoption and effective use of formal methods; (2) ***the lack of program repair methods*** specifically developed for programming scenarios such as those in robot EUD; (3) ***challenges within automated repair without human input*** for addressing scenarios where user feedback or input may not be feasible; and (4) ***the lack of human-centered design and understanding of tools and techniques*** that incorporate formal methods. Future research can feasibly address these challenges, thereby unleashing the potential of formal methods for robot EUD. We outline these gaps below.

7.2.1 Lack of Libraries, Implementations, and Best Practices

Program analysis, synthesis, and repair methods rely on task expectations, constraints that the systems must satisfy, and user preferences that all must be explicitly specified in machine-readable forms. While some specifications, such as user preferences, are highly user- and context-dependent, task, system, and environmental expectations and constraints can be generalizable across contexts. For example, Porfirio *et al.* (2018) specified a set of correctness properties for social norms in linear temporal logic (LTL; §2.2.1), which were then used to verify user programs for social norm violations. These norms, such as, that a robot must greet its user at first encounter, may be defined as reusable properties in libraries that can be shared and used across projects. Such libraries largely do not exist, and future efforts to create them can greatly ease the advancement and adoption of formal methods in robot EUD.

7.2.2 Need for Effective Repair Methods

As formal method techniques are increasingly applied to robot EUD, our review shows that, compared to creation and analysis, there is still limited exploration of program maintenance methods. Whereas studies on program repair are highly prevalent in areas related to software engineering and programming languages (*e.g.,* Li *et al.*, 2020; Le Goues *et al.*, 2019; Nguyen *et al.*, 2013), there isn't sufficient work done in the area of robot EUD. Interactive repair approaches that keep users in-the-loop are more common in human-robot HRI and robotics; still, even existing work is typically limited to domain-specific corrections and relies heavily on user input during execution (*e.g.,* Chung and Cakmak, 2020; Racca *et al.*, 2020). Few systems fully integrate repair into real-time, context-rich robot behaviors or support dynamic adaptation with formal guarantees. A potential approach to fostering the development of such methods is to engage the formal methods research community with technical and human-centered problems in robot EUD. For example, the development of "lightweight" formal methods (Kulik *et al.*, 2022; Jones *et al.*, 1996), specifically to support EUD can both advance robot EUD and promote the development of novel, more compact, and human-in-the-loop formal methods.

7.2.3 Challenges within Automated Repair without Human Input

As robots are deployed in everyday settings, situations and unexpected events that require repair will be inevitable. Robots will be placed in situations where they encounter unexpected events during runtime and scenarios that require different adaptation strategies based on user preferences or technical limitations (*e.g.,* Meng and Kress-Gazit, 2024; Chung and Cakmak, 2020). These scenarios require repair methods that go beyond the runtime, human-in-the-loop interaction schemes, and thus, automated repair methods will become essential for ensuring resilience in robot behavior. These methods must also include end users who can, for example, during design-time pre-define repair mechanisms that execute in response to failures or set deterministic boundaries or constraints that must not be violated when generating repaired programs. It is therefore important for future work to first acquire a

good understanding of what elements are needed to facilitate effective automated repair, and then explore the balance between automated repair and end-user control.

7.2.4 The Need to Incorporate User-centered Perspectives

The application of formal methods to robot EUD should also better incorporate human and user-centered perspectives. It is critical to understand how different types of users, including novices, domain experts, and expert developers, engage with EUD tools; how various stakeholders perceive and set expectations for these systems; and what limitations users may face when interacting with them. Our review highlights that much of the current work focuses on the development of technical methods to enable the application of formal methods to robot EUD. However, as these tools are ultimately meant for end users who create, analyze, and maintain robot programs, it is critical to consider their usability, accessibility, and acceptability. Such understanding will require examining how people actually use these tools, how useful they find them, and how systems can be designed to better align with user needs. As different phases of robot programming—namely analysis, creation, and maintenance—involve different end users, integrating these perspectives becomes essential for effective, real world adoption of these tools.

The design of EUD tools must also aim to increase usability, specifically in lowering the barrier for end users who must (1) define states and specifications, and (2) interpret system outputs and act upon them. Defining such states often requires translating user requests, preferences, and contextual constraints into formal representations such as verification properties or low-level logical operations. However, this translation process typically demands familiarity with formal languages—knowledge that everyday users often lack. Expecting users to manually perform or verify these translations poses a significant usability challenge. Practical solutions might handle much of this complexity "under the hood" where systems automatically and in real-time translate natural language requests into formal properties while maintaining accuracy and transparency. Advances in natural language processing (NLP) and large

language models (LLMs) can support this process; when properly constrained and grounded, the outputs of these LLMs can be used in formulating formal properties with more ease, making the system more approachable and responsive in everyday settings.

In addition to translating end-user input, robot EUD tools should also support user interaction and follow-up across different stages of the EUD process. Enabling users to understand, verify, and refine system outputs is critical for supporting iterative and meaningful engagement. For example, translating formal properties back into natural language, or providing visual explanations, can improve interpretability and transparency. In turn, systems can offer user control features that allow adjustments to constraints, such as their content, flexibility, or severity, based on evolving needs. These control mechanisms might involve natural language inputs (*e.g.,* Liu *et al.*, 2023; Tellex *et al.*, 2020), voice commands (*e.g.,* Porfirio *et al.*, 2023), visual interfaces (*e.g.,* Porfirio *et al.*, 2023; Lyons *et al.*, 2015; Lyons *et al.*, 2012), or other intuitive modalities. Hybrid approaches that combine formal verification with higher-level abstractions can help bridge the gap between system logic and user intent, ultimately lowering the barrier to entry and supporting the broader adoption of formal methods in EUD tools.

8

Practicum

This review has focused on characterizing the use of formal methods in robot end user development (EUD) as they appear in current literature. Another key goal of this work is to advocate for the use of these methods toward realizing their great potential for EUD and the development of robotics applications. To this end, this chapter aims to provide a practical guide for researchers aiming to integrate formal methods into robot EUD tools and interfaces.

In this chapter, we provide concrete guidelines for researchers interested in designing formal robot EUD tools and interfaces. Section §8.1 summarizes recurring patterns identified in prior work and provides heuristics for selecting representations and aligning formal techniques to these representations. Section §8.2 then discusses how formal methods can support each EUD phase in future work. We present three representative EUD workflows that include combinations of multiple EUD phases, illustrating potential applications through scenario-based examples. Lastly, section §8.3 makes a case for combining formal techniques with popular AI-based techniques, further providing insights on when to use formal techniques, AI-based techniques, or a combination of both.

8.1 From End-User Input to Formal Representations in Robot EUD

Existing work demonstrates that representations determine how end-user programs are interpreted, analyzed, and verified, and thus serve as the entry point to the formal methods pipeline. Choosing a representation dictates which formal techniques are applicable, and therefore what kinds of guarantees can be achieved. These guarantees include state-level guarantees based on invariant properties, transition-level guarantees involving pre-/postcondition checking, path-level guarantees derived from temporal logic properties, and existence guarantees that ensure the satisfaction of all—or as many as possible—constraints.

Despite their importance, selecting adequate representations remains a substantial challenge because end-user-written programs demonstrate large variance in their granularity, uncertainty, and structure. To provide guidance on tackling this challenge effectively, this section focuses on (1) the ***granularity of end-user input*** and (2) ***situational uncertainty and adaptivity***. Granularity of end-user input pertains to the level of detail that users must provide in their programs about robot behavior, such as step-by-step actions versus higher-level constraints. Situational uncertainty and adaptivity concern the extent to which user programs must accommodate future modifications and adaptations based on user preference evolution or task and environment variability.

8.1.1 Granularity of End-User Input

In EUD, two key paradigms tend to dominate—*imperative programming* and *declarative programming* (Fahland *et al.*, 2009). These popular paradigms are analogous to differing modes of human reasoning and the granularity of the instructions that serve as input to program creation. Van Roy and Haridi (2004) distinguish declarative programming from imperative programming as "[declarative programming] defining *what* (the results we want to achieve) without explaining the *how* (the algorithms, etc., needed to achieve results)".

From an HRI and robotics research perspective, the choice of representation reflects the amount of effort and the degree of step-by-step specification involved. Offering an EUD tool that supports such explicit,

procedural instruction—following that of the *imperative programming* paradigm—builds on users' domain-specific knowledge and procedural understanding of the task. This approach of data collection and processing is useful when users have expectations about the order of events that should occur within the interaction, such as assembling manufacturing parts in specific order (*e.g.,* Schoen *et al.*, 2020) or simulating a scenario where a robot greets customers that enter a store (*e.g.,* Porfirio *et al.*, 2018). Robot EUD approaches that accept granular user inputs usually leverage state-transition systems, such as finite state machines (FSMs; §2.2.1), as their intermediate representations. Existing approaches map each instruction step (*e.g.,* robot action) into FSM states and interpret the transitions between these states as the causal or temporal relations that guide program execution. With this mathematical formulation of end-user programs, different forms of reasoning become possible, affording guarantees such as whether certain properties hold in each state (*i.e.,* §4.2 *invariant correctness*; Hurnaus and Prähofer, 2010) and whether particular events occur or precede others along an execution path (*i.e.,* §4.2 *temporal correctness*; Askarpour *et al.*, 2021, Schoen *et al.*, 2020, Porfirio *et al.*, 2018).

On the other hand, EUD tools that support high-level, constraint-like specifications as user input aligns more closely with the *declarative programming* paradigm. Rather than requiring users to enumerate every step of execution, the declarative approach allows them to express desired goals, outcomes (*e.g.,* Kubota *et al.*, 2020), or relational constraints (*e.g.,* Gavran *et al.*, 2018) that the robot must satisfy. In theory, this abstraction can reduce the cognitive burden on users and enable greater flexibility in refining robot behaviors. Robot EUD tools that accept constraint-like user inputs usually leverage logical formulae, such as linear temporal logic (LTL) (§2.2.1), as their intermediate representations. This formulae summarize user intent within a set of verifiable conditions that govern program synthesis or guide the robot's autonomous behavior planning. With the declarative formulation of end-user intent, program guarantees on robot behaviors can be made through verifying that generated behaviors adhere to temporal, safety, and goal-oriented specifications. Such declarative representations not only ensure formal correctness but also facilitate adaptive behavior generation that remains

consistent with the user's intended outcomes across varying contexts.

8.1.2 Situational Uncertainty and Adaptivity

The level of input granularity determines not only the type of user specifications that can be expressed, but also *how* user intent can be effectively elicited, following the paradigms of imperative and declarative programming. In contrast, the level of situational uncertainty and adaptivity concerns *when* and *what* aspects of a task should be specified. Situational uncertainty and adaptivity arises in HRI and robotics because robots are frequently deployed in highly dynamic environments, making interactions between humans and robots inherently open-ended. Conducting research in these domains, and eliciting and developing contextually appropriate user specifications—deciding when to specify different levels of abstraction and how to support subsequent modifications—remains a non-trivial challenge.

In situations where robots should adhere to determined sequence of steps without much deviation, complete action sequences can be collected from the user. Using multiple end-user examples of robot action sequences, such as demonstration traces, synthesis techniques can identify recurring program structures (*e.g.,* loops or conditionals) and generate fully executable programs (*e.g.,* Porfirio *et al.*, 2023; Sauer and Henrich, 2022).

Conversely, in scenarios with greater uncertainty and the need for flexibility and adaptation—such as task-related failures, planned deviations from pre-defined robot capabilities, or interactions influenced by evolving user preferences are expected to arise at runtime—some degree of iterative modification becomes necessary. Logical formulae, such as LTL formulae, are often more amenable to such revision than stcenarios with greep-by-step action sequences, especially when the program complexity is high. Logical formulae can also be beneficial when the end user has confidence on what general modifications to apply across multiple demonstration traces. *Program sketches* (different from sketches in design) can also be useful, in which high-level structures are pre-defined and the remaining low-level details, such as parameter values, can be finalized via iterative tuning (*e.g.,* Chung and Cakmak,

2020; Racca *et al.*, 2020).

8.2 Integrating Formal Methods into Flexible EUD Workflows

Currently, many existing robot EUD tools equipped with formal method techniques apply one technique at a time. These tools operate on different assumptions and goals, such as either analyzing existing robot programs (*e.g.,* Porfirio *et al.*, 2018), creating robot programs from scratch (*e.g.,* Gavran *et al.*, 2020; Kubota *et al.*, 2020), or refining existing programs (*e.g.,* Meng and Kress-Gazit, 2024; Racca *et al.*, 2020). We argue that there are many benefits to supporting program analysis, creation, and maintenance activities in a *unified* manner, as these three activities revolve around a key, common concept—enforcing program *correctness*. To support this argument, we illustrate how formal method techniques—program verification, synthesis, and repair—can be integrated into each EUD phase, respectively, within the same tool. To demonstrate this idea, we outline three potential workflows that reflect different combinations of these EUD activities.

It is important to recognize that the three core EUD activities—analysis, creation, and maintenance—need not occur in a fixed sequence. In practice, robot EUD may have several entry points and life cycles: analyzing program building blocks before program creation, developing a program from scratch and then automatically maintaining it, and developing, analyzing, and automatically maintaining a program, to name a few examples. Reflecting this flexibility, we identify three workflows: (1) ***analysis to creation***, (2) ***creation to maintenance***, and (3) ***creation to analysis to maintenance.***

Analysis to Creation First, **analysis and creation** can be jointly supported within an EUD tool by verifying segments of end-user-written programs, such as traces or short sequences of robot actions that are subsequently used to construct full programs. Consider a scenario in which the user programs a robot to perform a pick-and-place operation. During the analysis phase, verification such as pre-/postcondition evaluation and type checking, may reveal that the "place" action has certain guarantees while also highlighting potential issues with this action.

For example, verification may indicate that if an object is not stably positioned, the place action will succeed with lower confidence. This instability can lead to failures in subsequent pick actions, which require that the object is stable enough to pick up. Using this information, the user can then assemble verifiable pick-and-place programs by manually assembling smaller verified segments or by synthesizing correct-by-construction programs (Kourie and Watson, 2012) from scratch based on the desired correctness properties.

Creation to Maintenance Second, **creation and maintenance** can be implemented within the same EUD tool by supporting end users synthesis of robot programs and their automatic, iterative refinement. Maintenance of existing programs becomes necessary when robot programs are adapted to different scenarios and user preferences. Building on the robot pick-and-place scenario, imagine a scenario where an end user automatically synthesized the pick-and-place task, and subsequently wants to adjust the parameters on the robot's gripping behavior. When running the existing robot program, the user may identify that the gripping force is insufficient for certain objects. During the maintenance phase, the user thereby needs to modify the gripper configuration so that it can handle heavier objects with an increased force value. These types of adjustments can be made by updating parameters (*e.g.,* Chung and Cakmak, 2020; Racca *et al.*, 2020) through SMT solving (SMT in §2.2.3), or re-synthesizing programs to exclude counterexamples (*e.g.,* Porfirio *et al.*, 2020) from the previously synthesized programs. This allows iterative improvements to the contextual fit and the robot's performance, without having to rebuild the entire program, including the unchanged program components.

Creation to Analysis to Maintenance Finally, all three core EUD activities—**creation, analysis, and maintenance**—can be simultaneously employed within the same robot EUD tool, where this comprehensive workflow enforces the correct construction, execution, and revision of correct robot behaviors over time. Consider the case where an end user constructs and verifies an existing program with respect to a set of existing LTL properties. Now, the user wants to modify the program

so that the robot performs pick-and-place with a human worker in a co-located space. During analysis, the end user may realize that the safety properties defined in the initial program are no longer adequate, as the robot can potentially collide with the human operator. The user thereby synthesizes a new set of linear temporal properties (LTL; §2.2.1) properties using the techniques discussed in §5.2. The user can then integrate these constraints into the existing program by utilizing the EUD tool's repair techniques, such as CEGIS (Solar-Lezama *et al.*, 2006), to include the updated safety requirements. The end result of this process is therefore a new, verified (*e.g.,* through model checking; §2.4) program that can continue to be maintained as safety properties evolve.

8.3 Hybrid Approaches to Robot EUD

Recent advances in robot EUD increasingly draw upon a range of AI-based techniques. While classical AI approaches, such as automated planners (*e.g.,* PDDL planners; Aeronautiques *et al.*, 1998), have often been used to achieve new goals given pre-defined action sets, the prevalence of machine learning/deep learning techniques, and large language model (LLM) applications have enabled a proliferation of robot EUD systems that build upon these capabilities.

The pervasiveness of AI and learning-related works in robot EUD reflects the importance of handling unforeseen cases and learning from limited data. However, a challenge persists in using AI to generalize end-user specifications in HRI and robotics beyond classic programming paradigms—safety boundaries and imposing task constraints are still paramount. In real world deployments, the failure to meet safety or performance expectations, especially in human-facing contexts, can cause costly and potentially irreparable consequences. Given the high cost required to deploy robots in the real world in addition to the cost for repairing robot failures, formally guaranteeing correctness and safety prior to deployment remains crucial.

In this subsection, we summarize current AI-based techniques in robot EUD—including automated planning, natural language programming, and other statistical machine learning methods (including

Bayesian and deep learning approaches)—and discuss how these techniques seek logical generalization. We then provide insights on how these techniques can be complemented by techniques in formal methods, thus providing "hybrid" approaches to robot EUD that both generalize and provide correctness guarantees.

Automated Planning Some robot EUD tools utilize automated planners to generate sequential action steps given robot action pre-/postconditions and one or more high-level goals (*e.g.,* Porfirio *et al.*, 2024; Liang *et al.*, 2022; Liang *et al.*, 2019; Gavran *et al.*, 2018). The primary benefit of using an automated planner is that it offloads the end-user burden of generating low-level action sequences, while at the same time ensuring that new action sequences can be generated for different tasks, environmental contexts, and users. EUD tools in this category primarily require users to provide high-level task specifications—such as desired end states, goals, or constraints—rather than manually enumerating step-by-step action sequences.

Natural Language Programming Because natural language reduces the burden of learning new specification formalisms and supports more expressive intent elicitation, many robot EUD tools take advantage of this expression modality, affording end users a relatively intuitive and flexible method for articulating robot programs (*e.g.,* Porfirio *et al.*, 2023; Gavran *et al.*, 2020; Porfirio *et al.*, 2019). An additional benefit of including natural language programming techniques is its constantly improving reliability—with LLMs, for example, great strides are continuously being made in correctly translating loosely specified natural language to fully executable programs (*e.g.,* Porfirio *et al.*, 2025; Karli *et al.*, 2024). In addition, improved contextual awareness is achieved through richer linguistic information, while also enhancing generalization by allowing EUD tools to interpret flexible, novel, or out-of-distribution end-user expressions.

Other Statistical Machine Learning Methods Robot EUD tools that use statistical machine learning techniques, including deep learning,

learn functions, policies, or distributions over user preferences, allowing these tools to generalize across diverse demonstrations and adapt to variability in user behavior (*e.g.,* Van Waveren *et al.*, 2022; Chung and Cakmak, 2020; Racca *et al.*, 2020; Mollard *et al.*, 2015). The benefits of using these techniques include capturing general trends and patterns in data without overfitting, while reducing the end-user burden for explicitly specifying mappings or relationships that are often difficult to precisely articulate. EUD tools in this category do not restrict users to using specific input formats: feedback may be explicit (*e.g.,* parameter differentials; Racca *et al.*, 2020) often consisting of prohibited actions Van Waveren *et al.*, 2022), implicit (*e.g.,* indicating actions to redo; Chung and Cakmak, 2020), or simply consist of demonstrations from which the system infers key events (*e.g.,* Mollard *et al.*, 2015).

When to Use Formal Methods, Learning, or Hybrid Approaches As we summarized the representative use cases of AI-based techniques in existing robot EUD work, we now focus on the ways in which these popular EUD approaches can integrate formal method techniques. Such integration combines the flexibility and adaptivity of AI-based techniques with the assurance and provable correctness of formal reasoning. We first identify the strengths and key characteristics of each category of technique.

AI-based techniques are useful when operating on continuous, high-dimensional data, and for handling variance across user inputs or task instances. Their strength lies in learning generalizable patterns and fitting trends rather than relying on specified constraints or logical formulae. Because task generalization is crucial in many HRI and robotics applications—and often requires integrating contextual information such as perception, language, or robot control data—the ability of AI-based approaches to fuse and reason over multimodal inputs provides a significant advantage.

Formal method techniques are particularly useful for expression and reasoning with discrete logics such as linear temporal logic (LTL; §2.2.1) and metric temporal logic (MTL; §2.2.1). Encoding end-user specifications in these formalisms enables the decomposition of program behavior and supports rigorous correctness reasoning at multiple levels—

state, path, and program. Formal method techniques are especially appropriate when absolute assurance of robot performance or program correctness is required (rather than merely high empirical accuracy), such as when enforcing hard safety constraints or task constraints that must be satisfied under all executions.

Given the complementary strengths of the aforementioned techniques, one could opt for utilizing hybrid approaches that combine both class of techniques appropriately. One instantiation of this might be to capture and use state-of-the-art AI synthesis techniques to generalize a full program from only a small number of user demonstrations, such as execution traces, while simultaneously leveraging formal verification to explicitly identify and enforce critical constraints. Further identification and iterative refinement of correctness criteria can be facilitated through runtime verification (RV; Leucker and Schallhart, 2009 and Kim *et al.*, 1999), which monitors executions online, detects deviations from specified properties, and informs subsequent updates to the learned programs or formal specifications. Utilizing such hybrid approaches lowers the barriers to integrating formal method techniques into existing research and implementation practices, enabling broader exploration of this synergy across diverse applications.

9

Conclusion

Robotic applications have the potential to be widely adopted in everyday scenarios. However, a challenge remains in helping end users navigate the complexity of creating and maintaining robot programs. This work surveys the unique combination of robot end user development (EUD) and formal method techniques that seeks to address this challenge. Combining formal methods with EUD aims to empower end users to generate and customize their own robot programs by offering them assistance throughout the program development cycle.

Motivated by the definition and operationalization of program correctness, this review maps each formal method to a particular phase within the EUD timespan. Verification techniques, which assess program correctness, are discussed in relation to the analysis phase within the EUD cycle. Synthesis techniques, which create provably correct programs, are discussed in relation to the creation phase. Lastly, repair techniques, which automatically refine and modify programs, are discussed in association with the maintenance EUD phase.

In order to provide clear guidelines for robot EUD researchers who are interested in applying formal techniques to their work, we structure the main chapters of our paper (§4 through §6) with regards to the

formal representations that one might use (*e.g.,* finite state machines and linear temporal logic formulae), how to map raw user input to these formal representations, and the specific formal method techniques that can be applied.

In later chapters, we argue that applying formal techniques to robot EUD can bring many benefits to the EUD research community, discuss the current research gaps where further work is needed, and propose research directions that can advance the integration of formal methods into EUD in human-robot interaction and HRI more broadly. Finally, we conclude this review with a practical guide for EUD researchers, offering a step-by-step list of action items and decision-making considerations for effectively incorporating formal techniques into robot programming workflows.

References

Aeronautiques, C., A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. Sri, A. Barrett, D. Christianson, *et al.* (1998). "Pddl| the planning domain definition language". *Technical Report, Tech. Rep.*

Ajaykumar, G., M. Steele, and C.-M. Huang. (2021). "A survey on end-user robot programming". *ACM Computing Surveys (CSUR).* 54(8): 1–36. DOI: 10.1145/3466819.

Akgun, B. and A. Thomaz. (2016). "Simultaneously learning actions and goals from demonstration". *Auton. Robots.* 40(2): 211–227. ISSN: 0929-5593. DOI: 10.1007/s10514-015-9448-x.

Alexandrova, S., Z. Tatlock, and M. Cakmak. (2015). "RoboFlow: A flow-based visual programming language for mobile manipulation tasks". In: *2015 IEEE International Conference on Robotics and Automation (ICRA).* IEEE. 5537–5544. DOI: 10.1109/icra.2015.7139973.

Alshiekh, M., R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. (2018). "Safe reinforcement learning via shielding". In: *Proceedings of the AAAI conference on artificial intelligence.* Vol. 32. No. 1. DOI: 10.1609/aaai.v32i1.11797.

Askarpour, M., L. Lestingi, S. Longoni, N. Iannacci, M. Rossi, and F. Vicentini. (2021). "Formally-based model-driven development of collaborative robotic applications". *Journal of Intelligent & Robotic Systems.* 102(3): 59. DOI: 10.1007/s10846-021-01386-2.

Audemard, G., P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. (2002). "A SAT based approach for solving formulas over boolean and linear mathematical propositions". In: *International Conference on Automated Deduction*. Springer. 195–210. DOI: 10.1007/3-540-45620-1_17.

Baader, F. (2003). *The description logic handbook: Theory, implementation and applications*. Cambridge university press. DOI: 10.1017/CBO9780511711787.

Baader, F., I. Horrocks, and U. Sattler. (2008). "Description logics". *Foundations of Artificial Intelligence*. 3: 135–179. DOI: 10.1016/S1574-6526(07)03003-9.

Baisero, A., Y. Mollard, M. Lopes, M. Toussaint, and I. Lütkebohle. (2015). "Temporal segmentation of pair-wise interaction phases in sequential manipulation demonstrations". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 478–484. DOI: 10.1109/iros.2015.7353415.

Barricelli, B. R., F. Cassano, D. Fogli, and A. Piccinno. (2019). "End-user development, end-user programming and end-user software engineering: A systematic mapping study". *Journal of Systems and Software*. 149: 101–137. DOI: 10.1016/j.jss.2018.11.041.

Beschi, S., D. Fogli, and F. Tampalini. (2019). "CAPIRCI: a multimodal system for collaborative robot programming". In: *End-User Development: 7th International Symposium, IS-EUD 2019, Hatfield, UK, July 10–12, 2019, Proceedings 7*. Springer. 51–66. DOI: 10.1007/978-3-030-24781-2_4.

Bjørner, N., A.-D. Phan, and L. Fleckenstein. (2015). "νz-an optimizing SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 194–199. DOI: 10.1007/978-3-662-46681-0_14.

Cagiltay, B., B. Mutlu, and J. E. Michaelis. (2023). ""My Unconditional Homework Buddy:" Exploring Children's Preferences for a Homework Companion Robot". In: *Proceedings of the 22nd Annual ACM Interaction Design and Children Conference*. 375–387. DOI: 10.1145/3585088.3589388.

Chen, Y. and G. De Luca. (2016). "VIPLE: visual IoT/robotics programming language environment for computer science education". In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 963–971. DOI: 10.1109/ipdpsw.2016.55.

Chen, Z. *et al.* (2003). "Bayesian filtering: From Kalman filters to particle filters, and beyond". *Statistics*. 182(1): 1–69. DOI: 10.1080/02331880309257.

Chidambaram, S., H. Huang, F. He, X. Qian, A. M. Villanueva, T. S. Redick, W. Stuerzlinger, and K. Ramani. (2021). "ProcessAR: An augmented reality-based tool to create in-situ procedural 2D/3D AR Instructions". In: *Proceedings of the 2021 ACM Designing Interactive Systems Conference. DIS '21*. Virtual Event, USA: Association for Computing Machinery. 234–249. ISBN: 9781450384766. DOI: 10.1145/3461778.3462126.

Chung, M. J.-Y. and M. Cakmak. (2020). "Iterative repair of social robot programs from implicit user feedback via bayesian inference". *interaction*. 1: 2. DOI: 10.15607/rss.2020.xvi.028.

Clarke, E. M., E. A. Emerson, and A. P. Sistla. (1986). "Automatic verification of finite-state concurrent systems using temporal logic specifications". *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 8(2): 244–263. DOI: 10.1145/5397.5399.

Clarke, E. M., T. A. Henzinger, H. Veith, R. Bloem, *et al.* (2018). *Handbook of model checking*. Vol. 10. Springer. DOI: 10.1007/978-3-319-10575-8.

Clarke, E. M., W. Klieber, M. Nováček, and P. Zuliani. (2011). "Model checking and the state explosion problem". In: *LASER Summer School on Software Engineering*. Springer. 1–30. DOI: 10.1007/978-3-642-35746-6_1.

Das, D., S. Banerjee, and S. Chernova. (2021). "Explainable ai for robot failures: Generating explanations that improve user assistance in fault recovery". In: *Proceedings of the 2021 ACM/IEEE international conference on human-robot interaction*. 351–360. DOI: 10.1145/3434073.3444657.

Datta, C., E. Broadbent, and B. A. MacDonald. (2016). "Formalizing the specifications of a domain-specific language for authoring behaviour of personal service robots". In: *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. IEEE. 98–103. DOI: 10.1109/simpar.2016.7862382.

De Moura, L. and N. Bjørner. (2008). "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

De Moura, L. and N. Bjørner. (2011). "Satisfiability modulo theories: introduction and applications". *Communications of the ACM*. 54(9): 69–77. DOI: 10.1145/1995376.1995394.

Dragan, A. and S. Srinivasa. (2014a). "Familiarization to robot motion". In: *Proceedings of the 2014 ACM/IEEE international conference on Human-robot interaction*. 366–373. DOI: 10.1145/2559636.2559674.

Dragan, A. and S. Srinivasa. (2014b). "Integrating human observer inferences into robot motion planning". *Autonomous Robots*. 37: 351–368. DOI: 10.1007/s10514-014-9408-x.

Dutertre, B. and L. De Moura. (2006). "A fast linear-arithmetic solver for DPLL (T)". In: *International Conference on Computer Aided Verification*. Springer. 81–94. DOI: 10.1007/11817963_11.

Ebbinghaus, H.-D., J. Flum, W. Thomas, and A. S. Ferebee. (1994). *Mathematical logic*. Vol. 2. Springer. DOI: 10.1007/978-1-4757-2355-7.

Ehlers, R. and V. Raman. (2016). "Slugs: Extensible gr (1) synthesis". In: *International Conference on Computer Aided Verification*. Springer. 333–339. DOI: 10.1007/978-3-319-41540-6_18.

Fahland, D., D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. (2009). "Declarative versus imperative process modeling languages: The issue of understandability". In: *International Workshop on Business Process Modeling, Development and Support*. Springer. 353–366. DOI: 10.1007/978-3-642-01862-6_29.

Fisac, J. F., M. A. Gates, J. B. Hamrick, C. Liu, D. Hadfield-Menell, M. Palaniappan, D. Malik, S. S. Sastry, T. L. Griffiths, and A. D. Dragan. (2020). "Pragmatic-pedagogic value alignment". In: *Robotics research: the 18th international symposium Isrr*. Springer. 49–57. DOI: 10.1007/978-3-030-28619-4_7.

Fogli, D., M. Peroni, and C. Stefini. (2017). "ImAtHome: Making trigger-action programming easy and fun". *Journal of Visual Languages Computing*. 42: 60–75. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2017.08.003.

Fox, M. and D. Long. (2003). "PDDL2. 1: An extension to PDDL for expressing temporal planning domains". *Journal of artificial intelligence research*. 20: 61–124. DOI: 10.1613/jair.1129.

Gavran, I., E. Darulova, and R. Majumdar. (2020). "Interactive synthesis of temporal specifications from examples and natural language". *Proceedings of the ACM on Programming Languages*. 4(OOPSLA): 1–26. DOI: 10.1145/3428269.

Gavran, I., O. Mailahn, R. Müller, R. Peifer, and D. Zufferey. (2018). "TOOL: accessible automated reasoning for human robot collaboration". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 44–56. DOI: 10.1145/3276954.3276961.

Goldschlager, L. and A. Lister. (1986). *Computer Science: A modern introduction*. Prentice Hall International (UK) Ltd. DOI: 10.1016/0898-1221(90)90200-4.

Gulwani, S. (2010). "Dimensions in program synthesis". In: *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. 13–24. DOI: 10.1145/1836089.1836091.

Gulwani, S. (2012). "Synthesis from examples". In: *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*. Vol. 10. No. 2. Citeseer.

Hagenow, M., E. Senft, R. Radwin, M. Gleicher, M. Zinn, and B. Mutlu. (2024). "A System for Human-Robot Teaming through End-User Programming and Shared Autonomy". In: *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*. 231–239. DOI: 10.1145/3610977.3634965.

Han, Z. and H. Yanco. (2023). "Communicating missing causal information to explain a robot's past behavior". *ACM Transactions on Human-Robot Interaction*. 12(1): 1–45. DOI: 10.1145/3568024.

Hasan, S., M. S. Yasar, and T. Iqbal. (2024). "M2RL: A Multimodal Multi-Interface Dataset for Robot Learning from Human Demonstrations". In: *Proceedings of the 26th International Conference on Multimodal Interaction. ICMI '24*. San Jose, Costa Rica: Association for Computing Machinery. 254–263. ISBN: 9798400704628. DOI: 10.1145/3678957.3685713.

Hoffmann, J. and B. Nebel. (2001). "The FF planning system: Fast plan generation through heuristic search". *Journal of Artificial Intelligence Research*. 14: 253–302. DOI: 10.1613/jair.855.

Holtz, J., A. Guha, and J. Biswas. (2018). "Interactive robot transition repair with SMT". *arXiv preprint arXiv:1802.01706*. DOI: 10.24963/ijcai.2018/681.

Holtz, J., A. Guha, and J. Biswas. (2020). "Smt-based robot transition repair". *arXiv preprint arXiv:2001.04397*. DOI: 10.48550/arXiv.2001.04397.

Hopcroft, J. E., R. Motwani, and J. D. Ullman. (2001). "Introduction to automata theory, languages, and computation". *Acm Sigact News*. 32(1): 60–65. DOI: 10.1016/0096-0551(80)90011-9.

Huang, G., P. S. Rao, M.-H. Wu, X. Qian, S. Y. Nof, K. Ramani, and A. J. Quinn. (2020). "Vipo: Spatial-visual programming with functions for robot-IoT workflows". In: *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–13. DOI: 10.1145/3313831.3376670.

Huang, J. and M. Cakmak. (2017). "Code3: A System for End-to-End Programming of Mobile Manipulator Robots for Novices and Experts". In: *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction. HRI '17*. Vienna, Austria: Association for Computing Machinery. 453–462. ISBN: 9781450343367. DOI: 10.1145/2909824.3020215.

Hurnaus, D. and H. Prähofer. (2010). "Programming assistance based on contracts and modular verification in the automation domain". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. 2544–2551. DOI: 10.1145/1774088.1774614.

Ikeda, B., M. Gramopadhye, L. Nekervis, and D. Szafir. (2025). "MARCER: Multimodal Augmented Reality for Composing and Executing Robot Tasks". In: *Proceedings of the 2025 ACM/IEEE International Conference on Human-Robot Interaction. HRI '25*. Melbourne, Australia: IEEE Press. 529–539. DOI: 10.1109/hri61500.2025.10974232.

Jha, S., S. Gulwani, S. A. Seshia, and A. Tiwari. (2010). "Oracle-guided component-based program synthesis". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 215–224. DOI: 10.1145/1806799.1806833.

Jiang, H., S. Elbaum, and C. Detweiler. (2017). "Inferring and monitoring invariants in robotic systems". *Autonomous Robots*. 41: 1027–1046. DOI: 10.1007/s10514-016-9576-y.

Jones, C. B., D. Jackson, and J. Wing. (1996). "Formal methods light". *Computer*. 29(04): 20–22. DOI: 10.1145/242224.242380.

Kalyan, A., A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. (2018). "Neural-guided deductive search for real-time program synthesis from examples". *arXiv preprint arXiv:1804.01186*. DOI: 10.48550/arXiv.1804.01186.

Karli, U. B., J.-T. Chen, V. N. Antony, and C.-M. Huang. (2024). "Alchemist: Llm-aided end-user development of robot applications". In: *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*. 361–370. DOI: 10.1145/3610977.3634969.

Kim, D., J. Nam, J. Song, and S. Kim. (2013). "Automatic patch generation learned from human-written patches". In: *2013 35th international conference on software engineering (ICSE)*. IEEE. 802–811. DOI: 10.1109/icse.2013.6606626.

Kim, M., M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. (1999). "Formally specified monitoring of temporal properties". In: *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*. IEEE. 114–122. DOI: 10.1109/emrts.1999.777457.

Ko, A. J., R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, *et al.* (2011). "The state of the art in end-user software engineering". *ACM Computing Surveys (CSUR)*. 43(3): 1–44. DOI: 10.1145/1922649.1922658.

Kourie, D. G. and B. W. Watson. (2012). *The Correctness-by-Construction Approach to Programming*. Vol. 264. Springer. DOI: 10.1007/978-3-642-27919-5.

Kovatsch, M., Y. N. Hassan, and S. Mayer. (2015). "Practical semantics for the Internet of Things: Physical states, device mashups, and open questions". In: *2015 5th International Conference on the Internet of Things (IOT)*. New York, NY, USA: IEEE. 54–61. DOI: 10.1109/IOT.2015.7356548.

Koymans, R. (1990). "Specifying real-time properties with metric temporal logic". *Real-time systems*. 2(4): 255–299. DOI: 10.1007/bf01995674.

Kress-Gazit, H., K. Eder, G. Hoffman, H. Admoni, B. Argall, R. Ehlers, C. Heckman, N. Jansen, R. Knepper, J. Křetínskỳ, *et al.* (2021). "Formalizing and guaranteeing human-robot interaction". *Communications of the ACM*. 64(9): 78–84. DOI: 10.1145/3433637.

Kress-Gazit, H., M. Lahijanian, and V. Raman. (2018). "Synthesis for robots: Guarantees and feedback for robot behavior". *Annual Review of Control, Robotics, and Autonomous Systems*. 1: 211–236. DOI: 10.1146/annurev-control-060117-104838.

Kubota, A., E. I. Peterson, V. Rajendren, H. Kress-Gazit, and L. D. Riek. (2020). "Jessie: Synthesizing social robot behaviors for personalized neurorehabilitation and beyond". In: *Proceedings of the 2020 ACM/IEEE international conference on human-robot interaction*. 121–130. DOI: 10.1145/3319502.3374836.

Kubota, A., M. Pourebadi, S. Banh, S. Kim, and L. Riek. (2021). "Somebody that i used to know: The risks of personalizing robots for dementia care". *Proceedings of We Robot*.

Kulik, T., B. Dongol, P. G. Larsen, H. D. Macedo, S. Schneider, P. W. Tran-Jørgensen, and J. Woodcock. (2022). "A survey of practical formal methods for security". *Formal aspects of computing*. 34(1): 1–39. DOI: 10.1145/3522582.

Kwiatkowska, M., G. Norman, and D. Parker. (2011). "PRISM 4.0: Verification of probabilistic real-time systems". In: *International conference on computer aided verification*. Springer. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

Kwon, M., S. H. Huang, and A. D. Dragan. (2018). "Expressing robot incapability". In: *Proceedings of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*. 87–95. DOI: 10.1145/3171221.3171276.

Lamport, L. (1977). "Proving the correctness of multiprocess programs". *IEEE transactions on software engineering*. (2): 125–143. DOI: 10.1109/tse.1977.229904.

Landay, J. and B. Myers. (2001). "Sketching interfaces: toward more human interface design". *Computer*. 34(3): 56–64. DOI: 10.1109/2.910894.

Le Goues, C., M. Pradel, and A. Roychoudhury. (2019). "Automated program repair". *Communications of the ACM*. 62(12): 56–65. DOI: 10.1145/3318162.

Lee, H. R. and L. Riek. (2023). "Designing robots for aging: Wisdom as a critical lens". *ACM Transactions on Human-Robot Interaction*. 12(1): 1–21. DOI: 10.1145/3549531.

Leonardi, N., M. Manca, F. Paternò, and C. Santoro. (2019). "Trigger-action programming for personalising humanoid robot behaviour". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–13. DOI: 10.1145/3290605.3300675.

Leucker, M. and C. Schallhart. (2009). "A brief account of runtime verification". *The journal of logic and algebraic programming*. 78(5): 293–303. DOI: 10.1016/j.jlap.2008.08.004.

Li, T. J.-J., A. Azaria, and B. A. Myers. (2017). "SUGILITE: Creating Multimodal Smartphone Automation by Demonstration". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. CHI '17*. Denver, Colorado, USA: Association for Computing Machinery. 6038–6049. DOI: 10.1145/3025453.3025483.

Li, T. J.-J., I. Labutov, B. A. Myers, A. Azaria, A. I. Rudnicky, and T. M. Mitchell. (2018). "Teaching Agents When They Fail: End User Development in Goal-Oriented Conversational Agents". In: *Studies in Conversational UX Design*. Ed. by R. J. Moore, M. H. Szymanski, R. Arar, and G.-J. Ren. Cham: Springer International Publishing. 119–137. ISBN: 978-3-319-95579-7. DOI: 10.1007/978-3-319-95579-7_6.

Li, T. J.-J., M. Radensky, J. Jia, K. Singarajah, T. M. Mitchell, and B. A. Myers. (2019). "PUMICE: A Multi-Modal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations". In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology. UIST '19*. New Orleans, LA, USA: Association for Computing Machinery. 577–589. DOI: 10.1145/3332165.3347899.

Li, Y., S. Wang, and T. N. Nguyen. (2020). "Dlfix: Context-based code transformation learning for automated program repair". In: *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 602–614. DOI: 10.1145/3377811.3380345.

Li, Z. and X. Si. (2022). "NSNet: A general neural probabilistic framework for satisfiability problems". *Advances in Neural Information Processing Systems*. 35: 25573–25585. DOI: 10.48550/arXiv.2211.03880.

Liang, Y. S., D. Pellier, H. Fiorino, and S. Pesty. (2019). "End-user programming of low-and high-level actions for robotic task planning". In: *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. IEEE. 1–8. DOI: 10.1109/roman46459.2019.8956327.

Liang, Y. S., D. Pellier, H. Fiorino, and S. Pesty. (2022). "iRoPro: An interactive robot programming framework". *International Journal of Social Robotics*: 1–15. DOI: 10.1007/s12369-021-00775-9.

Lieberman, H., F. Paternò, M. Klann, and V. Wulf. (2006). "End-user development: An emerging paradigm". In: *End user development*. Springer. 1–8. DOI: 10.1007/1-4020-5386-x_1.

Liu, J. X., Z. Yang, I. Idrees, S. Liang, B. Schornstein, S. Tellex, and A. Shah. (2023). "Grounding complex natural language commands for temporal tasks in unseen environments". In: *Conference on Robot Learning*. PMLR. 1084–1110. DOI: 10.48550/arXiv.2302.11649.

Liu, K., D. Sakamoto, M. Inami, and T. Igarashi. (2011). "Roboshop: multi-layered sketching interface for robot housework assignment and management". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '11*. Vancouver, BC, Canada: Association for Computing Machinery. 647–656. DOI: 10.1145/1978942.1979035.

Lou, Y., A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang. (2020). "Can automated program repair refine fault localization? a unified debugging approach". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87. DOI: 10.1145/3395363.3397351.

Lucci, N., A. Monguzzi, A. M. Zanchettin, and P. Rocco. (2022). "Workflow modelling for human–robot collaborative assembly operations". *Robotics and Computer-Integrated Manufacturing*. 78: 102384. DOI: 10.1016/j.rcim.2022.102384.

Lyons, D. M., R. C. Arkin, S. Jiang, D. Harrington, F. Tang, and P. Tang. (2015). "Probabilistic verification of multi-robot missions in uncertain environments". In: *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 56–63. DOI: 10.1109/ictai.2015.22.

Lyons, D. M., R. C. Arkin, P. Nirmal, and S. Jiang. (2012). "Designing autonomous robot missions with performance guarantees". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2583–2590. DOI: 10.1109/iros.2012.6385952.

Mandelin, D., L. Xu, R. Bodík, and D. Kimelman. (2005). "Jungloid mining: helping to navigate the API jungle". *ACM Sigplan Notices*. 40(6): 48–61. DOI: 10.1145/1064978.1065018.

Marques-Silva, J. (2008). "Practical applications of boolean satisfiability". In: *2008 9th International Workshop on Discrete Event Systems*. IEEE. 74–80. DOI: 10.1109/wodes.2008.4605925.

Matentzoglu, N., B. Parsia, R. S. Gonçalves, B. Glimm, and A. Steigmiller. (2017). "The OWL Reasoner Evaluation (ORE) 2015 Competition Report". *Journal of Automated Reasoning*. DOI: 10.1007/s10817-017-9406-8.

Mayer, S., R. Verborgh, M. Kovatsch, and F. Mattern. (2016). "Smart Configuration of Smart Environments". *IEEE Transactions on Automation Science and Engineering*. 13(3): 1247–1255. DOI: 10.1109/TASE.2016.2533321.

Meng, Q. and H. Kress-Gazit. (2024). "Automated Robot Recovery from Assumption Violations of High-Level Specifications". In: *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*. IEEE. 4154–4161. DOI: 10.1109/case59546.2024.10711578.

Mollard, Y., T. Munzer, A. Baisero, M. Toussaint, and M. Lopes. (2015). "Robot programming from demonstration, feedback and transfer". In: *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 1825–1831. DOI: 10.1109/iros.2015.7353615.

Moser, M., M. Pfeiffer, and J. Pichler. (2014). "A novel domain-specific language for the robot welding automation domain". In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE. 1–6. DOI: 10.1109/etfa.2014.7005348.

Myers, B. A. (1986). "Visual programming, programming by example, and program visualization: a taxonomy". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '86*. Boston, Massachusetts, USA: Association for Computing Machinery. 59–66. ISBN: 0897911806. DOI: 10.1145/22627.22349. URL: https://doi.org/10.1145/22627.22349.

Myers, B. A., J. F. Pane, and A. J. Ko. (2004). "Natural programming languages and environments". *Commun. ACM*. 47(9): 47–52. DOI: 10.1145/1015864.1015888.

Neider, D. (2014). "Applications of automata learning in verification and synthesis". *PhD thesis*. Aachen, Techn. Hochsch., Diss., 2014.

Nelson, C. G. (1980). *Techniques for program verification*. Stanford University.

Nguyen, H. D. T., D. Qi, A. Roychoudhury, and S. Chandra. (2013). "Semfix: Program repair via semantic analysis". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 772–781. DOI: 10.1109/icse.2013.6606623.

Norton, A., H. Admoni, J. Crandall, T. Fitzgerald, A. Gautam, M. Goodrich, A. Saretsky, M. Scheutz, R. Simmons, A. Steinfeld, *et al.* (2022). "Metrics for robot proficiency self-assessment and communication of proficiency in human-robot teams". *ACM Transactions on Human-Robot Interaction (THRI)*. 11(3): 1–38. DOI: 10.1145/3522579.

Noura, M., S. Heil, and M. Gaedke. (2018). "GrOWTH: Goal-Oriented End User Development for Web of Things Devices". In: *Web Engineering*. Ed. by T. Mikkonen, R. Klamma, and J. Hernández. Cham: Springer International Publishing. 358–365. DOI: https://doi.org/10.1007/978-3-319-91662-0_29.

Oney, S., B. Myers, and J. Brandt. (2014). "InterState: a language and environment for expressing interface behavior". In: *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology. UIST '14*. Honolulu, Hawaii, USA: Association for Computing Machinery. 263–272. DOI: 10.1145/2642918.2647358.

Ouaknine, J. and J. Worrell. (2008). "Some recent results in metric temporal logic". In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 1–13. DOI: 10.1007/978-3-540-85778-5_1.

Page, M. J., J. E. McKenzie, P. M. Bossuyt, I. Boutron, T. C. Hoffmann, C. D. Mulrow, L. Shamseer, J. M. Tetzlaff, E. A. Akl, S. E. Brennan, *et al.* (2021). "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". *bmj*. 372. DOI: 10.1136/bmj.n71.

Patton, N., K. Rahmani, M. Missula, J. Biswas, and I. Dillig. (2024). "Programming-by-Demonstration for Long-Horizon Robot Tasks". *Proc. ACM Program. Lang.* 8(POPL). DOI: 10.1145/3632860.

Pnueli, A. (1977). "The temporal logic of programs". In: *18th annual symposium on foundations of computer science (sfcs 1977)*. ieee. 46–57. DOI: 10.1109/sfcs.1977.32.

Porfirio, D., E. Fisher, A. Sauppé, A. Albarghouthi, and B. Mutlu. (2019). "Bodystorming human-robot interactions". In: *proceedings of the 32nd annual ACM symposium on user Interface software and technology*. 479–491. DOI: 10.1145/3332165.3347957.

Porfirio, D., V. Hsiao, M. Fine-Morris, L. Smith, and L. M. Hiatt. (2025). "Bootstrapping Human-Like Planning via LLMs". *arXiv preprint arXiv:2506.22604*. DOI: 10.48550/arXiv.2506.22604.

Porfirio, D., M. Roberts, and L. M. Hiatt. (2024). "Goal-Oriented End-User Programming of Robots". In: *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*. 582–591. DOI: 10.1145/3610977.3634974.

Porfirio, D., A. Sauppé, A. Albarghouthi, and B. Mutlu. (2018). "Authoring and verifying human-robot interactions". In: *Proceedings of the 31st annual acm symposium on user interface software and technology*. 75–86. DOI: 10.1145/3242587.3242634.

Porfirio, D., A. Sauppé, A. Albarghouthi, and B. Mutlu. (2020). "Transforming robot programs based on social context". In: *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–12. DOI: 10.1145/3313831.3376355.

Porfirio, D., L. Stegner, M. Cakmak, A. Sauppé, A. Albarghouthi, and B. Mutlu. (2023). "Sketching Robot Programs On the Fly". In: *Proceedings of the 2023 ACM/IEEE International Conference on Human-Robot Interaction*. 584–593. DOI: 10.1145/3568162.3576991.

Porfirio, D. J., L. Stegner, M. Cakmak, A. Sauppé, A. Albarghouthi, and B. Mutlu. (2021). "Figaro: A tabletop authoring environment for human-robot interaction". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15. DOI: 10.1145/3411764.3446864.

Pradella, M. (2009). "A User's Guide to Zot". *arXiv preprint arXiv:0912.5014*. DOI: 10.48550/arXiv.0912.5014.

Queille, J.-P. and J. Sifakis. (1982). "Specification and verification of concurrent systems in CESAR". In: *International Symposium on programming*. Springer. 337–351. DOI: 10.1007/3-540-11494-7_22.

Queille, J.-P. and J. Sifakis. (1983). "Fairness and related properties in transition systems—a temporal logic to deal with fairness". *Acta informatica*. 19: 195–220. DOI: 10.1007/bf00265555.

Racca, M., V. Kyrki, and M. Cakmak. (2020). "Interactive tuning of robot program parameters via expected divergence maximization". In: *Proceedings of the 2020 ACM/IEEE International Conference on Human-Robot Interaction*. 629–638. DOI: 10.1145/3319502.3374784.

Rahman, M. A., I. A. Felix, U. Shahid, and J. E. Michaelis. (2024). "PATHWiSE: An AI-Assisted Teacher Authoring Tool for Creating Custom Robot-Assisted Learning Activities". In: *Companion of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*. 88–90. DOI: 10.1145/3610978.3641086.

Raman, V. and H. Kress-Gazit. (2012). "Explaining impossible high-level robot behaviors". *IEEE Transactions on Robotics*. 29(1): 94–104. DOI: 10.1109/tro.2012.2214558.

Roy, N. and A. McCallum. (2001). "Toward optimal active learning through monte carlo estimation of error reduction". *Icml, williamstown*. 2(441-448): 4.

Sauer, L. and D. Henrich. (2022). "Structure synthesis for extended robot state automata". In: *International Conference on Robotics in Alpe-Adria Danube Region*. Springer. 71–79. DOI: 10.1007/978-3-031-04870-8_9.

Schenkenfelder, B., M. Moser, M. Pfeiffer, J. Pichler, C. Salomon, and M. Winterer. (2023). "Iterative Design and Evaluation of a Low-Code Development Platform for Welding Robot Control". In: *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 1–8. DOI: 10.1109/etfa54631.2023.10275606.

Schleicher, D., P. Jones, and O. Kachur. (2010). "Bodystorming as embodied designing". *interactions*. 17(6): 47–51. DOI: 10.1145/1865245.1865256.

Schoen, A., C. Henrichs, M. Strohkirch, and B. Mutlu. (2020). "Authr: A task authoring environment for human-robot teams". In: *Proceedings of the 33rd annual acm symposium on user interface software and technology*. 1194–1208. DOI: 10.1145/3379337.3415872.

Senft, E., M. Hagenow, R. Radwin, M. Zinn, M. Gleicher, and B. Mutlu. (2021a). "Situated live programming for human-robot collaboration". In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. 613–625. DOI: 10.1145/3472749.3474773.

Senft, E., M. Hagenow, K. Welsh, R. Radwin, M. Zinn, M. Gleicher, and B. Mutlu. (2021b). "Task-level authoring for remote robot teleoperation". *Frontiers in Robotics and AI*. 8: 707149. DOI: 10.3389/frobt.2021.707149.

Settles, B. (2009). "Active learning literature survey".

Short, E. S., A. Allevato, and A. L. Thomaz. (2019). "SAIL: Simulation-Informed Active In-the-Wild Learning". In: *2019 14th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. 468–477. DOI: 10.1109/HRI.2019.8673019.

Sipser, M. (2013). “Regular Languages”. In: *Introduction to the Theory of Computation*. Cengage Learning. 31–47.

Solar-Lezama, A. (2008). *Program synthesis by sketching*. University of California, Berkeley.

Solar-Lezama, A., L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. (2006). “Combinatorial sketching for finite programs”. In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415. DOI: 10.1145/1168918.1168907.

Steffen, B., F. Howar, and M. Merten. (2011). “Introduction to active automata learning from a practical perspective”. *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures 11*: 256–296. DOI: 10.1007/978-3-642-21455-4_8.

Stegner, L. and B. Mutlu. (2022). “Designing for caregiving: Integrating robotic assistance in senior living communities”. In: *Proceedings of the 2022 ACM Designing Interactive Systems Conference*. 1934–1947. DOI: 10.1145/3532106.3533536.

Tellex, S., N. Gopalan, H. Kress-Gazit, and C. Matuszek. (2020). “Robots that use language”. *Annual Review of Control, Robotics, and Autonomous Systems*. 3: 25–55. DOI: 10.1146/annurev-control-101119-071628.

Thati, P. and G. Roşu. (2005). “Monitoring algorithms for metric temporal logic specifications”. *Electronic Notes in Theoretical Computer Science*. 113: 145–162. DOI: 10.1016/j.entcs.2004.01.029.

Vaiani, G. and F. Paternò. (2024). “End-User Development for Human-Robot Interaction: Results and Trends in an Emerging Field”. *Proceedings of the ACM on Human-Computer Interaction*. 8(EICS): 1–40. DOI: 10.1145/3661146.

Van Roy, P. and S. Haridi. (2004). *Concepts, techniques, and models of computer programming*. MIT press.

Van Waveren, S., C. Pek, J. Tumova, and I. Leite. (2022). "Correct me if I'm wrong: Using non-experts to repair reinforcement learning policies". In: *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE. 493–501. DOI: 10.1109/hri53351.2022.9889604.

Wang, T., X. Qian, F. He, X. Hu, Y. Cao, and K. Ramani. (2021). "GesturAR: An Authoring System for Creating Freehand Interactive Augmented Reality Applications". In: *The 34th Annual ACM Symposium on User Interface Software and Technology. UIST '21*. Virtual Event, USA: Association for Computing Machinery. 552–567. DOI: 10.1145/3472749.3474769.

Wang, T., X. Qian, F. He, X. Hu, K. Huo, Y. Cao, and K. Ramani. (2020). "CAPturAR: An Augmented Reality Tool for Authoring Human-Involved Context-Aware Applications". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology. UIST '20*. Virtual Event, USA: Association for Computing Machinery. 328–341. DOI: 10.1145/3379337.3415815.

Wing, J. M. (1990). "A Specifier's Introduction to Formal Methods". *Computer*. 23(9): 8–23. ISSN: 0018-9162. DOI: 10.1109/2.58215. URL: https://doi.org/10.1109/2.58215.

Wong, K. W. and H. Kress-Gazit. (2017). "From high-level task specification to robot operating system (ros) implementation". In: *2017 First IEEE International Conference on Robotic Computing (IRC)*. IEEE. 188–195. DOI: 10.1109/irc.2017.18.

Zhang, L., W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur. (2019). "AutoTap: Synthesizing and Repairing Trigger-Action Programs Using LTL Properties". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 281–291. DOI: 10.1109/ICSE.2019.00043.

Zhang, Q., C. Fang, Y. Ma, W. Sun, and Z. Chen. (2023). "A survey of learning-based automated program repair". *ACM Transactions on Software Engineering and Methodology*. 33(2): 1–69. DOI: 10.1145/3631974.