Reactive High-level Behavior Synthesis for an ATLAS Humanoid Robot

Spyros Maniatopoulos*, Philipp Schillinger†, Vitchyr H. Pong*, David C. Conner‡, and Hadas Kress-Gazit*

***Abstract*— In this work, we take a step towards bridging the gap between the theory of formal synthesis and its application to real-world, complex, robotic systems. In particular, we present an *end-to-end* approach for the automatic generation of code that implements high-level robot behaviors in a verifiably-correct manner, including reaction to the possible *failures* of low-level actions. We start with a description of the system defined *a priori*. Thus, a non-expert user need only specify a high-level task. We automatically construct a formal specification in a fragment of Linear Temporal Logic (LTL) that encodes the system's capabilities and constraints, the task, and the desired reaction to low-level failures. We then synthesize a *reactive* mission plan that is guaranteed to satisfy the formal specification, i.e., achieve the task's goals or correctly react to failures. Finally, we automatically generate a state machine that instantiates the synthesized symbolic plan in software.**

We showcase our approach using Team ViGIR's software and Atlas humanoid robot and present lab experiments, thus demonstrating the application of formal synthesis techniques to complex robotic systems. The proposed approach has been implemented and open-sourced as a collection of Robot Operating System (ROS) packages, which are adaptable to other systems.

I. Introduction

In preparation for the 2015 DARPA Robotics Challenge (DRC) Finals, Team ViGIR, as well as many other teams, developed an approach to high-level robot control [1]. However, these approaches relied on experts developing scripted behaviors or, in our case, manually designing state-machine-based controllers. In addition, there was no guarantee that the resulting high-level behaviors were correct with respect to the tasks at hand. Moreover, many participants observed that such approaches were fragile in practice [2]. Motivated by these shortcomings, and enabled by recent advances in the field of formal methods for robotics [3]–[17], we present an approach for the synthesis of provably correct reactive high-level robot behavior and the automatic generation of the corresponding software implementation.

Example 1: Consider the DRC task of walking to the valve and turning it (Fig. 1). Carrying out this task involves steps such as object detection, footstep planning, manipulation, etc. These capabilities also differ from robot to robot. Ideally, a non-expert user should be able to specify such tasks in terms of the goals to be achieved, without having to worry about the details of how the robotic system will achieve them. Moreover, it would be advantageous if any failures that might occur during execution were automatically handled.

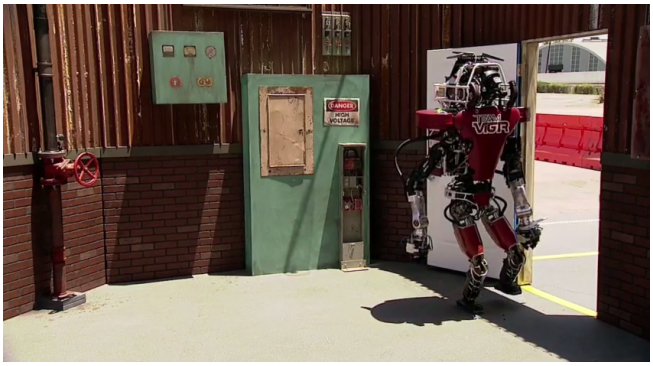


Fig. 1: Team ViGIR's Atlas humanoid robot on the first day of the DRC Finals. (Photo credit: DARPA)

To this end, we automatically generate logic-based specifications from a higher level, multi-paradigm specification: a user-specified task and the description of the system's capabilities, which includes constraints such as action preconditions, and is set up *a priori* by the system designers. Furthermore, most approaches employing formal methods assume that the low-level system components that make up the high-level plan will work as expected, i.e., they never fail. In this paper, we lift this assumption by formally accounting for the possibility of failure when executing the low-level components. While there might be no way to recover from such a failure, we can still achieve *graceful degradation*. That is, we can formally specify the system's reaction to failure.

Finally, we bridge the gap between theoretical results and practice. Thus, we automatically generate the software implementation of a state machine that instantiates the synthesized mission plan by grounding it to the robotic system's capabilities. The result is in an end-to-end approach. We present and experimentally validate this approach in the context of a complex system, a Boston Dynamics Atlas humanoid robot (Fig. 1) running the software that Team ViGIR developed for the DRC. Note that the concepts are general and apply to different systems. We have implemented and open-sourced the proposed approach as a collection of Robot Operating System (ROS [18]) packages.

The rest of this paper is organized as follows. In Section

This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Air Force Research Lab (AFRL) contract FA8750-12-C-0337.

The authors are with Team ViGIR, http://www.teamvigir.org

*Verifiable Robotics Research Group, Sibley School of Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY 14853, USA {sm2296, vhp22, hadaskg}@cornell.edu

†Robert Bosch GmbH, Corporate Research, Department for Cognitive Systems, 70442 Stuttgart, Germany philipp.schillinger@de.bosch.com

‡Capable Humanitarian Robotics & Intelligent Systems Lab, Department of Physics, Computer Science and Engineering, Christopher Newport University, Newport News, VA 23606, USA david.conner@cnu.edu

II, we compare our contributions to the state of the art. In Section III, we introduce Atlas, Team ViGIR's approach to the DRC, and Linear Temporal Logic. In Section IV, we state the technical problem that this paper addresses. We present our approach in Sections V through VII and summarize our ROS implementation in Section VIII. We present experimental demonstrations in Section IX. We draw conclusions and propose future work in Section X.

II. Related Work

We opted for automatically constructing the formal specification from user input and an *a priori* description of the system. Another option would have been asking the user to write the entire formal specification directly, e.g., in LTL, Structured English [5], or natural language [6]. Filippidis, et al. [19] also employ synthesis from multi-paradigm specifications. The imperative element of their specification language could have benefited our work, but is not necessary.

In terms of prior efforts to achieve graceful degradation in formal synthesis, Lahijanian, et al. [7] and Kim and Fainekos [10] consider the partial satisfaction of a formal specification, if it cannot be completely achieved. Conversely, Guo and Dimarogonas [11] guarantee the satisfaction of the hard portion of a specification and gradually improve that of the soft portion. DeCastro, et al. [12] deal with the problem of uncertainty in mobile robot motion. Johnson and Kress-Gazit [13] propose a probabilistic framework for analyzing sensor and actuation errors in formal synthesis. By contrast, we can account for, and specify a reaction to, worst-case failures (and other outcomes) of a real-world system's components.

In terms of the mission planning step, we opted for GR(1) synthesis [20], i.e., reactive LTL synthesis, over other approaches. These included classical AI planners, such as STRIPS [21] and PDDL [22], optimization under LTL constraints [14], and synthesis from co-safe LTL specifications [7]–[9]. Our main reason for choosing GR(1) synthesis is the ability to specify reactivity with respect to a dynamic, and even adversarial (worst-case), environment, such as external events sensed by the robot and low-level system failures.

Mehta, et al. [15] also present an end-to-end approach (from formal specification to code generation), while the toolkit developed by Finucane, et al. [16] employs an executive that executes the synthesized symbolic automaton without the need for code generation. In our work, the user does not have to input a complete specification. Most importantly, our approach takes advantage of existing robotics software, namely, ROS [18] and its large number of packages.

Finally, we *experimentally* validate our work on a real, complex system, Atlas, whereas most works in formal methods for robotics are only demonstrated in simulation.

III. Preliminaries

A. Atlas Humanoid Robot

Atlas (Fig. 1) is an anthropomorphic robot developed by Boston Dynamics, Inc. (BDI). Team ViGIR chose to leverage the basic capabilities provided by the Boston Dynamics Application Programming Interface (API). In the context of this work, we are especially interested in BDI's "control mode" interface (Fig. 2). The active control mode dictates which joints are controlled by the low-level BDI controllers and which joints we can command. For example, in STAND and MANIPULATE BDI's software handles balancing.

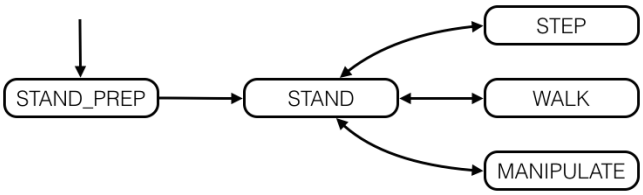


Fig. 2: Excerpt from the BDI control mode interface. Some changes between modes (depicted as arrows) are unidirectional while others are bidirectional. After Team ViGIR's initial checkout, Atlas is in the STAND_PREP control mode.

B. Team ViGIR's Approach to the DRC Finals

Team ViGIR has based its software on the ROS. In this section, we highlight some elements of the software's design that are relevant to high-level control and synthesis. For a complete overview of Team ViGIR's approach, we refer the interested reader to [23]. From this point on, we will refer to Atlas running Team ViGIR's software as the *system*, $\mathcal{S}$.

Since Team ViGIR uses BDI's control mode interface, some system capabilities are preconditioned on a certain control mode being active. For example, in order to execute an arm trajectory, the system must be in the MANIPULATE mode. In addition, the system's operation has to respect the constraints on the possible control mode changes (Fig. 2). We call such considerations system-specific safety requirements.

In terms of manipulation, Team ViGIR employs the concept of object templates [24]. In brief, the system presents the human operator with perception data, e.g., a point cloud. Then, the operator detects objects of interest and overlays an object template on top of them. These templates contain metadata, such as relative robot poses from which the object is reachable, relative pre-grasp and grasp end-effector poses, as well as finger configurations corresponding to different grasps. In addition, object templates provide manipulation affordances. For instance, the "door" template provides affordances such as "turn (handle) clockwise" and "push".

High-level Control: Team ViGIR's approach to high-level control is especially relevant to this work. Its corner stone is the Flexible Behavior Engine[1] (FlexBE [1]), which is a major extension of the SMACH high-level executive [25].

Using the FlexBE framework, developers create "state implementations", Q. Each $q \in Q$ is a small, atomic block of code that interfaces with *one* of the lower-level system capabilities $\mathcal{C}$. Furthermore, each state implementation defines a number of outcomes $Out(q)$, e.g., $\{\texttt{done}, \texttt{failed}, \texttt{aborted}\}$. The state implementations can be composed to form hierarchical state machines,[2] which encode the logic of execution as well as the flow of data

[1] https://github.com/team-vigir/flexbe_behavior_engine
[2] https://github.com/team-vigir/vigir_behaviors

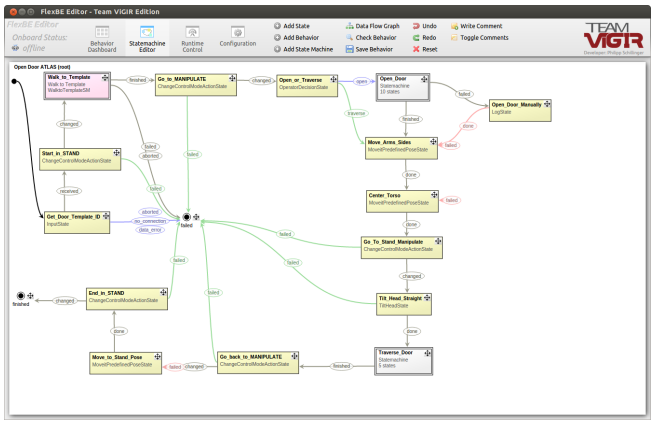


Fig. 3: A manually designed state machine for carrying out the DRC Finals' "Door" task. The initial state is indicated by the black arrow originating from the top left. The state machine has two outcomes, "finished" (bottom left) and "failed" (center). Yellow states are parametrized state implementations. Gray and purple states are state machines. In brief, the high-level behavior implemented by this state machine is one where the system first asks the human operator to identify the door handle and then Atlas approaches the door, turns the handle, and steps through the door.

(Fig. 3). Specifically, state machines consist of *parametrized* instantiations, $q_p \in Q_P$, of the state implementations Q. For example, if a state implementation corresponds to changing control modes, its parametrized counterparts correspond to changing to specific control modes, e.g., to STAND. State machines also have outcomes themselves, e.g. $\{\texttt{finished}, \texttt{failed}\}$. Finally, FlexBE state machines can be reused by saving and then embedding them in other state machines, e.g., state `Walk_to_Template` (purple) in Fig. 3.

Both composition of state machines and supervision of their execution takes place in FlexBE's graphical user interface[3] (GUI). Figure 3 depicts an example of a state machine that implements a high-level behavior (opening and traversing through a door). It was designed manually in the FlexBE GUI's editor by an expert user.

C. Linear Temporal Logic and Reactive LTL Synthesis

Linear Temporal Logic (LTL) is a formal language that combines Boolean ($\neg$, $\wedge$, $\vee$) and temporal (next $\bigcirc$, until $\mathcal{U}$) operators. Additional temporal operators, always $\Box$, eventually $\Diamond$, can be derived from those. LTL formulas are constructed from Boolean atomic propositions $\pi \in AP$. In the context of our work, the set of atomic propositions, AP, consists of propositions controlled by the system, $\mathcal{Y}$, and propositions controlled by the dynamic, and possibly adversarial, environment, $\mathcal{X}$. That is, $AP = \mathcal{X} \cup \mathcal{Y}$.

In order to synthesize *reactive* mission plans in a computationally tractable manner, we use the GR(1) fragment of LTL [20]. GR(1) formulas φ have an assume-guarantee structure between the dynamic environment (e) and the system (s):

$$\begin{aligned} \varphi &= (\varphi_e \Rightarrow \varphi_s), \\ \varphi_e &= \varphi_e^i \wedge \varphi_e^t \wedge \varphi_e^g, \\ \varphi_s &= \varphi_s^i \wedge \varphi_s^t \wedge \varphi_s^g, \end{aligned} \tag{1}$$

where the superscript i denotes initial conditions, t safety assumptions/requirements, and g liveness assumptions/requirements (i.e., goals) for e and s, respectively.

GR(1) synthesis involves setting up a two-player game between e and s [20]. If a GR(1) specification φ is realizable for s, we can extract a finite-state automaton; specifically, a Mealy machine. This automaton encodes a strategy for s that guarantees φ_s for any evolution of e that satisfies φ_e.

IV. Problem Statement & Approach Outline

Problem 1 (High-level Behavior Synthesis): Given a system $\mathcal{S}$, which comprises a robot and its primitive capabilities $\mathcal{C}$, a system designer-specified reaction to failures $\mathcal{F}$, and a user-provided task $\mathcal{T}$, in terms of its goals $\mathcal{G}$ and initial conditions[4] $\mathcal{I}$, automatically generate the software implementation of a high-level plan that *guarantees* the system will either:

i achieve the goals $\mathcal{G}$ safely, if possible,
ii or react according to the specification $\mathcal{F}$, if the execution of any primitive capabilities $\mathcal{C}$ does not succeed.

Problem 1 involves three main considerations. First, we have to accept input, $\mathcal{T}$, from non-expert users. In addition, we have to formally specify the task to be carried out by the system in order to capture the goals and initial conditions, system-specific safety requirements, and the possibility of failure. This formal specification should also account for the desired system's reaction to failure $\mathcal{F}$. Finally, we want to automatically generate a high-level plan that is verifiably-correct with respect to the formal specification, as well as its software implementation. These considerations give rise to the following subproblem statements, which together outline our approach to tackling Problem 1.

Problem 2 (Discrete Abstraction): Given Atlas' control mode transition constraints and the available actions $\mathcal{A}$, define a discrete abstraction $\mathcal{D}$ of the robot-plus-software system, $\mathcal{S}$, that captures both the execution and possible outcomes of its primitive capabilities (control mode transitions and actions). In addition, maintain a mapping, $\gamma : \mathcal{D} \to \mathcal{C}$, which grounds[5] the abstract symbols of the discrete abstraction to the primitive capabilities $\mathcal{C}$ of the system $\mathcal{S}$.

Problem 3 (Formal Task Specification): Given a task $\mathcal{T}$ in terms of goals $\mathcal{G} = \{g_1, g_2, \ldots, g_n\}$, the task's initial conditions $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$, the desired reaction to failures $\mathcal{F}$, and the discrete abstraction $\mathcal{D}$ of the system $\mathcal{S}$ that is to carry out the task,[6] automatically generate a specification $\mathcal{T}_\mathcal{S}$ that encodes, in a formal language, the task being carried out by $\mathcal{S}$.

[3] https://github.com/team-vigir/flexbe_chrome_app

[4] If the task specification process is taking place online, i.e., during system operation, then the initial conditions could also be detected automatically.

[5] We refer the reader to [26] for a review of symbol grounding in robotics.

[6] In general, a discrete abstraction of the robot's workspace would also be an input to this problem. However, we are not modeling it explicitly here.

To illustrate these concepts, consider again the scenario in Example 1. We could say that the task's goals $\mathcal{G}$ are $\{\texttt{turn\_valve}\}$ and its initial conditions $\mathcal{I}$ are, e.g., $\{\texttt{stand}\}$. The system $\mathcal{S}$ is Atlas running Team ViGIR's software, as described in Section III-B. Its primitive capabilities $\mathcal{C}$ include walking, rotating the wrist joint, and closing the fingers. The system-specific safety requirements (e.g., action preconditions and control mode changes) are encoded in $\mathcal{D}$.

Problem 4 (Mission Plan & Software Implementation): Given a formal task specification $\mathcal{T}_\mathcal{S}$, synthesize a reactive mission plan that is guaranteed to satisfy $\mathcal{T}_\mathcal{S}$, if one exists. Additionally, if such a plan exists, and given the mapping γ, automatically generate a software implementation of the reactive mission plan.

We present our approach to Problems 2, 3, and 4 in Sections V, VI, and VII, respectively.

V. Discrete Abstraction

A. Control Modes & Actions

We model Atlas' control mode interface (Fig. 2) as a transition system $(\mathcal{M}, \rightarrow)$, where $\mathcal{M}$ is the set of states, each corresponding to one control mode, $m \in \mathcal{M}$, and $\rightarrow$ is a set of valid control mode transitions (subset of $\mathcal{M} \times \mathcal{M}$). In addition, we define $Adj(m) = \{m' \in \mathcal{M} \mid (m, m') \in \rightarrow\}$ and also allow self-transitions, i.e., $m \in Adj(m)$, $\forall m \in \mathcal{M}$.

Furthermore, the system can perform actions. Actions, $a \in \mathcal{A}$, correspond to system capabilities $\mathcal{C}$ (other than control mode changes), e.g., generation of a footstep plan or closing the fingers. Actions may also have one or more preconditions. The preconditions of an action can be control modes or (the prior completion of) other actions, i.e., $Prec(a) \in 2^{(\mathcal{A} \cup \mathcal{M})}$ and $a \notin Prec(a)$, $\forall a \in \mathcal{A}$.

B. Atomic Propositions

We adopt a paradigm that generalizes the one in [3]. We abstract the actions, $a \in \mathcal{A}$, that Atlas can perform (activate) using one system proposition, π_a, per action and one environment proposition, π_a^o, per possible outcome of that action, $o \in Out(a)$. Similarly, for each control mode, $m \in \mathcal{M}$, we also have an activation proposition π_m and a number of outcome propositions π_m^o. Therefore, the set of atomic propositions AP is given by Eq. (2):

$$\mathcal{Y} = \bigcup_{a \in \mathcal{A}} \pi_a \cup \bigcup_{m \in \mathcal{M}} \pi_m, \tag{2a}$$

$$\mathcal{X} = \bigcup_{a \in \mathcal{A}} \bigcup_{o \in Out(a)} \pi_a^o \cup \bigcup_{m \in \mathcal{M}} \bigcup_{o \in Out(m)} \pi_m^o, \tag{2b}$$

It is up to the system designer to decide which possible outcomes of activating a system capability should be modeled explicitly. For the sake of simplicity of presentation, here we abstract all positive outcomes as "completion" (c) and all negative outcomes as "failure" (f). That is, $Out(y) = \{c, f\}$, $\forall y \in \mathcal{A} \cup \mathcal{M}$.

We say (somewhat informally) that the resulting discrete abstraction $\mathcal{D}$ consists of symbols ($m \in \mathcal{M}$, $a \in \mathcal{A}$, $\pi \in AP$) and relations between them (e.g., $\rightarrow$, $Prec$, Out). We defer a discussion of the mapping $\gamma : \mathcal{D} \rightarrow \mathcal{C}$ until Section VII.

VI. Formal Task Specification

A. Multi-Paradigm Specification

Specifying a robot task in a formal language can be a time consuming and error prone process. It also requires an expert user. To alleviate these issues, we employ a multi-paradigm specification approach. We first observe that there are portions of the task specification $\mathcal{T}_\mathcal{S}$ that are going to be system-specific and portions that are going to be task-specific, such as the task's goals. Intuitively, a non-expert user should only have to specify the goals $\mathcal{G}$ without worrying about the internals of the robot and the software it is running. We can infer which actions $\mathcal{A}$ are pertinent to a task and use the discrete abstraction $\mathcal{D}$ as the basis for automatically generating the portion of the formal specification that is related to the system itself. Finally, the initial conditions $\mathcal{I}$ are either specified by the user or detected at runtime.

Thus, referring to Problem 3, we get the goals $\mathcal{G}$ and initial conditions $\mathcal{I}$ from a user. The discrete abstraction $\mathcal{D}$ is system-specific and has been defined *a priori* by the expert system designers, according to Section V. We can now automatically generate the task specification $\mathcal{T}_\mathcal{S}$ in (the GR(1) fragment [20] of) Linear Temporal Logic. Since LTL is compositional, we can generate individual formulas and then conjunct them to get the full LTL specification.

B. Specification of Actions and Control Mode Constraints

Since the activation of capabilities is controlled by the system, the corresponding LTL formulas will be in φ_s, the safety requirements (see Section III-C). Conversely, we do not control the outcome of activation; the adversarial environment does. Therefore, the LTL formulas specifying the behavior of outcomes will be in φ_e, the safety assumptions.

1) General Formulas: We say that an activation proposition π_y, $y \in \{a, m\}$, is `True` when the corresponding primitive capability is being activated and `False` when it is not being activated[7]. Therefore, the system safety requirement (3) dictates that all activation propositions $\pi_y \in \mathcal{Y}$ should turn `False` once an outcome has been returned. Note that the left-hand side of formula (3) is only `True` at those distinct time steps where an outcome was just returned.

$$\bigwedge_{o \in Out(y)} \square \left(\pi_y \wedge \bigcirc \pi_y^o \Rightarrow \bigcirc \neg \pi_y \right) \tag{3}$$

The environment safety assumption (4) dictates that the outcomes, π_y^o, of the activation of any system capability are mutually exclusive (e.g., an action cannot both succeed and fail). Formula (4) also allows for no outcome being `True`.

$$\bigwedge_{o \in Out(y)} \square \left(\bigcirc \pi_y^o \Rightarrow \bigwedge_{o' \neq o} \bigcirc \neg \pi_y^{o'} \right) \tag{4}$$

[7]Note that this is in contrast to the work of Raman, et al. [3], where, e.g., π_{camera} being `False` stands for the act of *deactivating* the corresponding primitive capability, i.e., turning a camera off.

The environment safety assumption (5) constraints the value of outcomes. Specifically, it dictates that, if an outcome is `False` and the corresponding capability is not being activated, then that outcome should remain `False`. It is a generalization of formula (4) in [3].

$$\bigwedge_{o \in Out(y)} \square \left(\neg \pi_y^o \wedge \neg \pi_y \Rightarrow \bigcirc \neg \pi_y^o \right) \tag{5}$$

2) Action-specific Formulas: We now encode the connection between the activation and the possible outcomes of the system's actions, $a \in \mathcal{A}$. The environment safety assumption (6) dictates that the value of an outcome should not change if the corresponding action has not been activated again. In other words, outcomes persist through time.

$$\bigwedge_{o \in Out(a)} \square \left(\pi_a^o \wedge \neg \pi_a \Rightarrow \bigcirc \pi_a^o \right) \tag{6}$$

The environment liveness assumption (7) is a fairness condition. It states that (always) eventually, the activation of an action will result in an outcome, unless that action is not activated to begin with.

$$\square \diamondsuit \left(\left(\pi_a \wedge \bigvee_{o \in Out(a)} \bigcirc \pi_a^o \right) \vee \neg \pi_a \right) \tag{7}$$

The system safety requirement (8) constrains the activation of an action a unless its preconditions, $Prec(a)$, are met.

$$\square \Big(\bigvee_{y \in Prec(a)} \neg \pi_y^c \Rightarrow \neg \pi_a \Big) \tag{8}$$

where the superscript $c \in Out(y)$ stands for "completion".

3) Control Mode Formulas: For brevity of notation, let

$$\varphi_m = \pi_m \wedge \bigwedge_{m' \neq m} \neg \pi_{m'}$$

Activating φ_m, as opposed to π_m, takes into account the mutual exclusion between control modes $m \in \mathcal{M}$. Also let

$$\varphi_{\mathcal{M}}^{none} = \bigwedge_{m \in \mathcal{M}} \neg \pi_m,$$

where $\varphi_{\mathcal{M}}^{none}$ being `True` stands for not activating any control mode transitions, i.e., staying in the same control mode.

The system safety requirement (9) encodes the BDI control mode transition system (Section V-A, Fig. 2) in LTL.

$$\bigwedge_{m \in \mathcal{M}} \square \left(\bigcirc \pi_m^c \Rightarrow \bigvee_{m' \in Adj(m)} \bigcirc \varphi_{m'} \vee \bigcirc \varphi_{\mathcal{M}}^{none} \right) \tag{9}$$

The environment safety assumption (10) enforces mutual exclusion between the BDI control modes.

$$\bigwedge_{m \in \mathcal{M}} \square \left(\bigcirc \pi_m^c \Leftrightarrow \bigwedge_{m' \neq m} \bigcirc \neg \pi_{m'}^c \right) \tag{10}$$

The environment safety assumption (11) governs how the active control mode can change (or not) in a single time step, in response to the activation of a control mode transition.

$$\bigwedge_{m \in \mathcal{M}} \bigwedge_{m' \in Adj(m)} \square \left(\pi_m^c \wedge \varphi_{m'} \Rightarrow \left(\bigcirc \pi_m^c \bigvee_{o \in Out(m')} \bigcirc \pi_{m'}^o \right) \right) \tag{11}$$

Similar to (6), the environment safety assumption (12) dictates that the value of the outcomes of control mode transitions must not change if no transition is being activated.

$$\bigwedge_{m \in \mathcal{M}} \bigwedge_{o \in Out(m)} \square \left(\pi_m^o \wedge \varphi_{\mathcal{M}}^{none} \Rightarrow \bigcirc \pi_m^o \right) \tag{12}$$

The environment liveness assumption (13) is the equivalent of the fairness condition (7) for control modes. A single formula suffices for mutually exclusive propositions [3].

$$\square \diamondsuit \Big(\bigvee_{m \in \mathcal{M}} \left(\varphi_m \wedge \bigvee_{o \in Out(m)} \bigcirc \pi_m^o \right) \vee \varphi_{\mathcal{M}}^{none} \Big) \tag{13}$$

This concludes the system-specific portion of $\mathcal{T}_{\mathcal{S}}$.

C. Specification of Task Goals

Motivated by the DRC tasks, we present formulas that encode the accomplishment of each goal once. However, LTL can naturally handle repeating tasks (e.g. patrolling). We can even combine the two paradigms, e.g., "Accomplish the goals $\mathcal{G}$ infinitely often, but if anything fails, abort".

The system safety requirements (14) - (16) and liveness requirement (17) specify the achievement of the user-provided goals, $g \in \mathcal{G}$, over a finite run (using the same LTL semantics as for infinite execution). In this paradigm, we say that the execution itself has outcomes too. We denote them by $o \in Out(Exec)$ and, for simplicity, $Out(Exec) = \{c, f\}$. Note that the propositions corresponding to these outcomes, π_{Exec}^o, are system, not environment, propositions. We also introduce auxiliary system propositions, μ_g, which serve as memory of having accomplished each goal $g \in \mathcal{G}$.

$$\bigwedge_{g \in \mathcal{G}} \square \left(\bigcirc \pi_g^c \vee \mu_g \Leftrightarrow \bigcirc \mu_g \right) \tag{14}$$

$$\square \left(\pi_{Exec}^c \Leftrightarrow \bigwedge_{g \in \mathcal{G}} \mu_g \right) \tag{15a}$$

$$\square \left(\pi_{Exec}^f \Leftrightarrow \bigvee_{\pi \in \mathcal{Y}} \pi^f \right) \tag{15b}$$

$$\bigwedge_{o \in Out(Exec)} \square \left(\pi_{Exec}^o \Rightarrow \bigcirc \pi_{Exec}^o \right) \tag{16}$$

$$\square \diamondsuit \Big(\bigvee_{o \in Out(Exec)} \pi_{Exec}^o \Big) \tag{17}$$

The formulas above can be interpreted as: "If nothing fails, then eventually accomplish each goal. Otherwise, abort". That is, we assume that the desired reaction to failure $\mathcal{F}$ in Problem 3 is to stop execution. While this may sound simplistic and overly conservative, it is actually in line with real-world settings. For example, NASA JPL's Mars rovers automatically terminate an autonomous drive if the activation

of any actuators results in excessive motor current, rover tilt, wheel slip, etc [27]. Of course this is but one option; the system designers can specify different reactions to failure.

Formula (14) does not guarantee that the goals will be achieved in a specific order. However, that is often desirable. To this end, we can define the goals as an ordered set $\mathcal{G} = \{g_1, g_2, \ldots, g_n\}$, where $g_i < g_j$ for $i < j$, and the relation $g_i < g_j$ means that goal g_i has to be achieved before g_j. With this definition, we can replace the safety requirement (14) with (18), whenever strict goal order is desired.

$$\bigwedge_{i=1}^{n} \square \left((\pi_{g_i} \wedge \bigcirc \pi^c_{g_i}) \wedge \mu_{g_{i-1}} \vee \mu_{g_i} \Leftrightarrow \bigcirc \mu_{g_i} \right), \qquad (18)$$

where $\mu_{g_0} \triangleq \texttt{True}$. Formula (18) forces the system to carry out goal g_i after it has accomplished goal g_{i-1}. It can still activate the capability corresponding to π_{g_i} earlier, as necessitated by other parts of the task, but that will not count towards achievement of g_i (indicated by μ_{g_i} being `True`).

Finally, these auxiliary propositions (memory and outcomes of the run) are added to the system propositions:

$$\mathcal{Y} = \mathcal{Y} \cup \bigcup_{g \in \mathcal{G}} \mu_g \cup \bigcup_{o \in Out(Exec)} \pi^o_{Exec}$$

D. Specification of Initial Conditions

For each action a and control mode m in the initial conditions $\mathcal{I}$, the completion proposition should be `True` in the environment initial conditions (19). All other outcome propositions should be `False`.

$$\varphi^e_i = \bigwedge_{i \in \mathcal{I}} \left(\pi^c_i \bigwedge_{o \in Out(i) \setminus \{c\}} \neg \pi^o_i \right) \wedge \bigwedge_{j \notin \mathcal{I}} \bigwedge_{o \in Out(j)} \neg \pi^o_j \qquad (19)$$

Activation propositions are `False` regardless of whether that capability is in $\mathcal{I}$ or not because, if something is already an initial condition, then the resulting plan should not activate it at the beginning of execution. The auxiliary propositions are also `False`. Essentially, all $\pi \in \mathcal{Y}$ are initially `False`:

$$\varphi^s_i = \bigwedge_{i \in \mathcal{I}} \neg \pi_i \wedge \bigwedge_{j \notin \mathcal{I}} \neg \pi_j \bigwedge_{g \in \mathcal{G}} \neg \mu_g \bigwedge_{o \in Out(Exec)} \neg \pi^o_{Exec} \qquad (20)$$

VII. Reactive Mission Planning & Instantiation

We tackle Problem 4 in two sequential steps. First, we automatically generate a correct-by-construction symbolic automaton from the formal specification $\mathcal{T}_S$ using GR(1) synthesis (Section III-C, [20]). Specifically, we use the tool `slugs` [28], which implements a synthesis algorithm introduced by Raman, et al. [3], [4]. Their algorithm can handle a fragment of LTL slightly larger than GR(1). Namely, the one that includes $\bigcirc$ (next) operators in liveness formulas, such as formulas (7) and (13) in Section VI. If the formal specification, $\mathcal{T}_S$, is realizable, we can extract a finite-state automaton. An example is provided in Fig. 4.

The computational complexity of GR(1) synthesis is quadratic in the size of the state space [20], which in turn is exponential in the number of atomic propositions (state explosion problem). Thus, the introduction of outcome and memory propositions might raise concerns. However, the state space increase is far from worst-case exponential, because outcome propositions are tied to the corresponding activation propositions via the environment safety assumptions (Section VI). Likewise, memory propositions are highly constrained by formulas (14) or (18). For these reasons, an exponential blow-up is mitigated in practice.

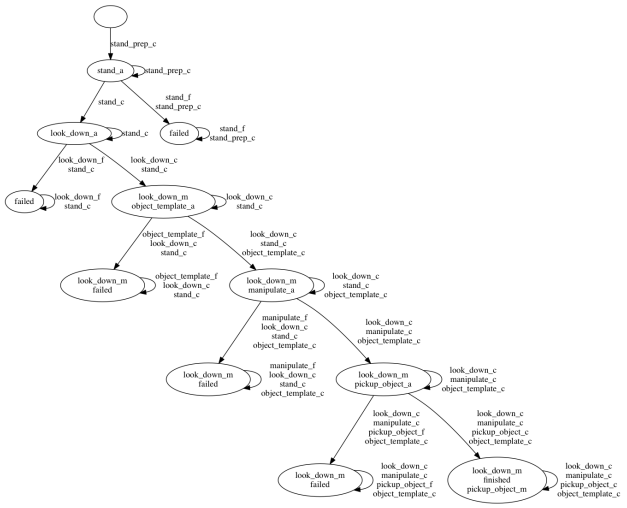


Fig. 4: The output of GR(1) synthesis is a symbolic finite-state automaton. For clarity, only the atomic propositions that are `True` are depicted. The formal specification was generated from the user input $\mathcal{I} = \{\texttt{stand\_prep}\}$ and $\mathcal{G} = \{\texttt{look\_down}, \texttt{pickup\_object}\}$, according to Section VI. We will revisit this example in Section VII. In general, the synthesized automata are not tree-like structures. This is an artifact of our choice of formulas in Section VI-C.

Second, we use the mapping $\gamma : \mathcal{D} \to \mathcal{C}$ to automatically generate code that instantiates the symbolic automaton. Without loss of generality, we generate executable state machines in the FlexBE framework introduced in Section III-B. Thus, the primitive system capabilities $\mathcal{C}$ are invoked via the execution of parametrized FlexBE states Q_P.

Specifically, activation propositions $\pi_y \in \mathcal{Y}$, $y \in \{a, m\}$, that evaluate to `True` in a state of the synthesized automaton are instantiated as a state $q_p \in Q_P$ in the FlexBE state machine. Furthermore, the corresponding outcome propositions $\pi^o_y \in \mathcal{X}$, $o \in Out(y)$, are mapped to the outcomes of that state, $Out(q_p)$. In practice, an outcome proposition can correspond to multiple outcomes of the state implementation. The auxiliary memory propositions introduced in Section VI-C were only necessary for the specification and synthesis steps. They do not have to be instantiated in software. Finally, in the finite run case, the outcome propositions $\pi^o_{Exec} \in \mathcal{Y}$, $o \in Out(Exec)$, are mapped to the outcomes of the FlexBE state machine, $Out(SM)$. The transitions of the synthesized automaton are mapped to those of the state machine. Examples of FlexBE state machines generated using our behavior synthesis approach are provided in Section IX, e.g., Fig. 6a.

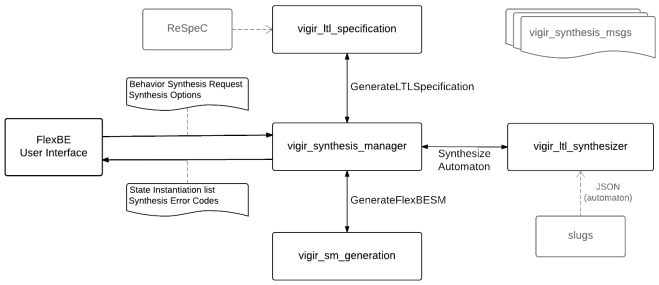


Fig. 5: Team ViGIR's "Behavior Synthesis" ROS packages and the nominal workflow (clockwise, starting from the left).

VIII. ROS Implementation

We have implemented all aspects of our approach in `vigir_behavior_synthesis`,[8] a collection of Robot Operating System (ROS) Python packages (Fig. 5).

The synthesis action server (`vigir_synthesis_manager`) receives a request from the user via FlexBE's GUI. Given the user's input (initial conditions and goals), the server first requests a full set of LTL formulas for Atlas from the `GenerateLTLSpecification` service (`vigir_ltl_specification` package). The generation of the LTL formulas from Section VI is delegated to our "Reactive Specification Construction kit" (ReSpeC),[9] which is a Python framework with rudimentary ROS integration.

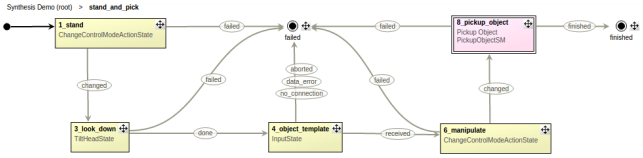
The `vigir_ltl_synthesizer` package acts as a wrapper for external synthesis tools (currently, `slugs` [28] is supported). Given the generated LTL specification, the `SynthesizeAutomaton` service returns a finite-state automaton that is guaranteed to satisfy it, if one exists. Lastly, the server requests a `StateInstantiation` message from the `GenerateFlexBESM` service (`vigir_sm_generation` package). This message provides the FlexBE Editor with sufficient information to generate Python code, i.e., an executable state machine that instantiates the synthesized automaton. The corresponding action, services, and messages are defined in the `vigir_synthesis_msgs` package.

IX. Experimental Validation

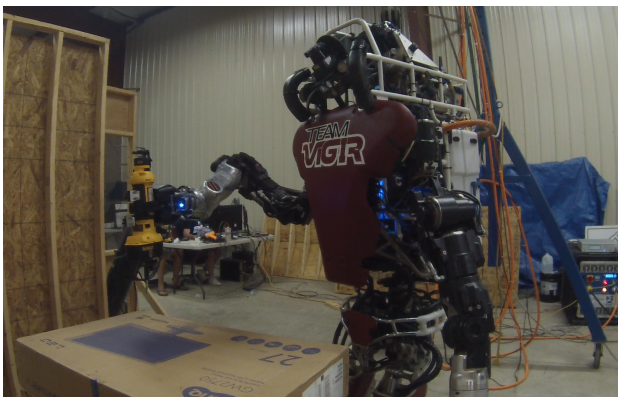
We carried out experimental demonstrations on Atlas in the lab, where, due to a hardware issue, Atlas could not locomote. In the first experimental demo, we show how a high-level behavior is specified and synthesized starting from scratch. Once the state machine has been instantiated (Fig. 6a), it is ready for execution (Fig. 6b).

For the second experimental demonstration, consider a scenario where the operator has designed a state machine that addresses a high-level task (either manually or via synthesis). Atlas is then deployed and starts carrying out this task. If, during execution, an *unexpected* situation arises, the operator can use FlexBE's runtime modification capability [1] (Fig. 7). In this case, behavior execution is "locked" at some state, i.e., this state is prevented from returning an outcome (Fig. 7a). Then, the operator specifies a new high-level behavior

[8] https://github.com/team-vigir/vigir_behavior_synthesis
[9] https://github.com/team-vigir/ReSpeC



(a) The state machine above was synthesized for the task with $\mathcal{I} = \{\texttt{stand\_prep}\}$ and $\mathcal{G} = \{\texttt{look\_down}, \texttt{pickup\_object}\}$. The capability `object_template` requests an object template from the operator (Section III-B). It is a precondition of `pickup_object`.



(b) Atlas finishing execution of state `8_pickup_object`.

Fig. 6: Snapshots from the first experimental demonstration.

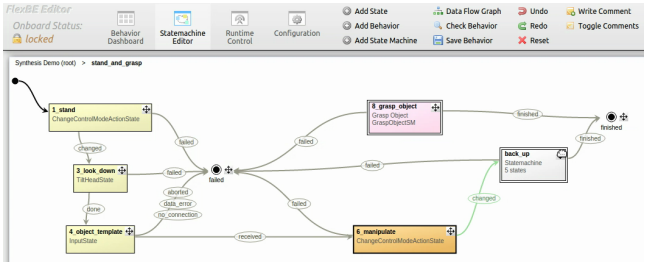
meant to address the unexpected situation. Once this new state machine is instantiated (Fig. 7b), it is connected to the previous one (Fig. 7a), and execution resumes.

X. Discussion and Future Work

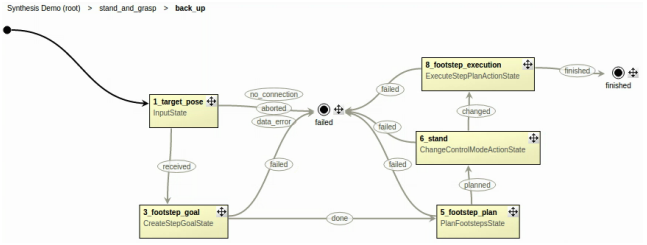
We presented an end-to-end approach to mission planning for complex robotic systems. We combined a task, specified by a non-expert user, with a discrete abstraction of the system, defined *a priori*, to automatically generate a formal specification. We then synthesized a provably correct mission plan that achieves the task's goals or reacts to any failures in the low-level system components. Finally, we automatically generated a software implementation of the mission plan in the form of an executable state machine. We implemented our approach as a collection of ROS packages and experimentally validated it on an Atlas humanoid robot running the software that Team ViGIR developed for the DRC.

It is important to point out the trade-off between expressivity and automation. On the one hand, an expert user can manually write a very expressive and customized formal specification. On the other hand, the generation of the formal specification can be automated, as we do here, but possibly at the expense of expressivity (e.g., due to hard-coded design choices.) However, there is no research barrier to recovering expressivity; the formal language supports it. Thus, we plan on extending our user interface in immediate future work.

The discrete abstraction and formal specification paradigm that we presented in this paper constitute the first steps towards achieving graceful degradation. In other words, we hinted at the question, "What formal guarantees can we offer when the execution of robot capabilities can result in failure?" We plan on further exploring this research direction.



(a) The operator "locks" the initial state machine at the state `6_manipulate` (indicated by the orange color), which is allowed to be executed. Then, a new state machine, `back_up`, is synthesized with `manipulate` as the initial condition. The transition from `6_manipulate` is then moved from `8_grasp_object` to `back_up`.



(b) The new state machine, `back_up`, was synthesized for the task with $\mathcal{I} = \{\texttt{manipulate}\}$ and $\mathcal{G} = \{\texttt{footstep\_execution}\}$.

Fig. 7: FlexBE Editor snapshots from the second experimental demonstration. In response to some event, the operator synthesized a state machine that has Atlas back away (7b).

We are also interested in automating the construction of the discrete abstraction, which includes action preconditions, outcomes, etc. Currently, the system designer defines it (once per system). We believe that, by formally specifying the capabilities and constraints of individual system components, we will be able to automatically generate the discrete abstraction. Finally, we will be demonstrating our approach on a number of different robotic systems in the near future.

Acknowledgments

The authors thank all other members of Team ViGIR and especially Alberto Romay, Stefan Kohlbrecher, and Prof. Oskar von Stryk from Technische Universität Darmstadt.

References

[1] P. Schillinger, "An Approach for Runtime-Modifiable Behavior Control of Humanoid Rescue Robots," Master's thesis, Technische Universität Darmstadt, 2015.

[2] DRC Teams, "What happened at the DARPA Robotics Challenge?" 2015. [Online]. Available: http://www.cs.cmu.edu/~cga/drc/events

[3] V. Raman, N. Piterman, and H. Kress-Gazit, "Provably Correct Continuous Control for High-Level Robot Behaviors with Actions of Arbitrary Execution Durations," in *IEEE Int'l. Conf. on Robotics and Automation*, 2013.

[4] V. Raman, N. Piterman, C. Finucane, and H. Kress-Gazit, "Timing semantics for abstraction and execution of synthesized high-level robot control," *IEEE Transactions on Robotics*, vol. 31, no. 3, pp. 591–604, 2015.

[5] G. Jing, C. Finucane, V. Raman, and H. Kress-Gazit, "Correct high-level robot control from structured english," in *IEEE International Conference on Robotics and Automation*, 2012, pp. 3543–3544.

[6] C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit, "Provably correct reactive control from natural language," *Autonomous Robots*, vol. 38, no. 1, pp. 89–105, 2015.

[7] M. Lahijanian, S. Almagor, D. Fried, L. E. Kavraki, and M. Y. Vardi, "This time the robot settles for a cost: A quantitative approach to temporal logic planning with partial satisfaction," in *The Twenty-Ninth AAAI Conference*, Austin, TX, January 2015, pp. 3664–3671.

[8] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, "Towards manipulation planning with temporal logic specifications," in *IEEE Intl. Conf. Robotics and Automation*, Seattle, WA, May 2015, pp. 346–352.

[9] E. Aydin Gol, M. Lazar, and C. Belta, "Language-guided controller synthesis for linear systems," *Automatic Control, IEEE Transactions on*, vol. 59, no. 5, pp. 1163–1176, May 2014.

[10] K. Kim and G. Fainekos, "Revision of specification automata under quantitative preferences," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, May 2014, pp. 5339–5344.

[11] M. Guo and D. Dimarogonas, "Distributed plan reconfiguration via knowledge transfer in multi-agent systems under local ltl specifications," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, May 2014, pp. 4304–4309.

[12] J. A. DeCastro, V. Raman, and H. Kress-Gazit, "Dynamics-driven adaptive abstraction for reactive high-level mission and motion planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, Seattle, WA, May 2015, pp. 369–376.

[13] B. Johnson and H. Kress-Gazit, "Analyzing and revising synthesized controllers for robots with sensing and actuation errors," *I. J. Robotic Res.*, vol. 34, p. 816–832, 2015.

[14] E. Wolff, U. Topcu, and R. Murray, "Optimization-based trajectory generation with linear temporal logic specifications," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, May 2014, pp. 5319–5325.

[15] A. Mehta, J. DelPreto, K. W. Wong, S. Hamill, H. Kress-Gazit, and D. Rus, "Robot creation from functional specifications," in *Int'l. Symposium on Robotics Research*, Sestri Levante, Italy, Sept 2015.

[16] C. Finucane, G. Jing, and H. Kress-Gazit, "LTLMoP: Experimenting with language, temporal logic and robot control," in *IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems*, Oct. 2010, pp. 1988–1993.

[17] H. Kress-Gazit, T. Wongpiromsarn, and U. Topcu, "Correct, reactive, high-level robot control," *Robotics Automation Magazine, IEEE*, vol. 18, no. 3, pp. 65–74, Sept 2011.

[18] Robot Operating System (ROS). [Online]. Available: www.ros.org

[19] I. Filippidis, R. M. Murray, and G. J. Holzmann, "A multi-paradigm language for reactive synthesis," in *4th Workshop on Synthesis (SYNT'15)*, ser. Electronic Proceedings in Theoretical Computer Science (EPTCS), San Francisco, CA, USA, July 2015.

[20] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911 – 938, 2012.

[21] R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, pp. 189 – 208, 1971.

[22] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL – The Planning Domain Definition Language," Yale Center for Computational Vision and Control, Tech. Rep., October 1998.

[23] S. Kohlbrecher, A. Romay, A. Stumpf, A. Gupta, O. von Stryk, F. Bacim, D. Bowman, A. Goins, R. Balasubramanian, and D. Conner, "Human-Robot Teaming for Rescue Missions: Team ViGIR's Approach to the 2013 DARPA Robotics Challenge Trials," *Journal of Field Robotics*, vol. 32, no. 3, pp. 352–377, 2015.

[24] A. Romay, S. Kohlbrecher, D. Conner, A. Stumpf, and O. von Stryk, "Template-Based Manipulation in Unstructured Environments for Supervised Semi-Autonomous Humanoid Robots," in *IEEE-RAS Intl. Conf. Humanoid Robots*, Madrid, Spain, Nov 2014, pp. 979–986.

[25] J. Bohren and S. Cousins, "The SMACH High-Level Executive [ROS News]," *Robotics Automation Magazine, IEEE*, vol. 17, no. 4, pp. 18–20, Dec 2010.

[26] S. Coradeschi, A. Loutfi, and B. Wrede, "A short review of symbol grounding in robotic and intelligent systems," *KI*, vol. 27, no. 2, pp. 129–136, 2013.

[27] J. Biesiadecki and M. Maimone, "The mars exploration rover surface mobility flight software: Driving ambition," in *IEEE Aerospace Conference*, 2006.

[28] R. Ehlers, V. Raman, and C. Finucane. (2013) Slugs GR(1) synthesizer. [Online]. Available: https://github.com/VerifiableRobotics/slugs