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 paper we present an *end-to-end* approach for the automatic generation of software that implements reactive high-level robot behaviors in a verifiably-correct manner. Our approach starts with an informal description of the system, as well as a task's goals and initial conditions. First, we abstract the problem and automatically construct a formal task specification in a fragment of Linear Temporal Logic (LTL). One highlight of our formalism is that it accounts for the possible failures of the various system components. We then synthesize a *reactive* mission plan that is guaranteed to satisfy the formal specification. Finally, we automatically generate a state machine that instantiates the synthesized plan in software.**

Specifically, this paper focuses on Team ViGIR's software and Atlas humanoid robot. The proposed approach has been implemented and open-sourced as a collection of Robot Operating System (ROS) packages. We demonstrate the efficacy of our approach via experiments with Atlas in the lab.

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], [2]. However, these approaches relied on experts developing scripted behaviors or, in the case of Team ViGIR, manually designing state machines. In addition, there was no guarantee that the resulting high-level behavior was correct with respect to (w.r.t.) the task at hand. Moreover, many participants observed that such approaches were fragile in practice [3].

Motivated by these shortcomings, we present an approach for the automatic generation of software that implements high-level robot behaviors in a provably-correct manner. This was enabled in part by recent advances in the field of formal methods for robotics. Specifically, correct-by-construction mission plans can be automatically synthesized from high-level, logic-based specifications [cite ALL the papers].

Example 1: Consider the task, "Walk to the valve and turn it" (Fig. 1). This would be an intuitive way to express the task from a non-expert user's point-of-view. However, this task specification is not formal, it does not account for the robot's capabilities – a robot with no means of locomotion or manipulation wouldn't even be able to carry it out – and it does not specify what should happen if a failure occurs.

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

*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

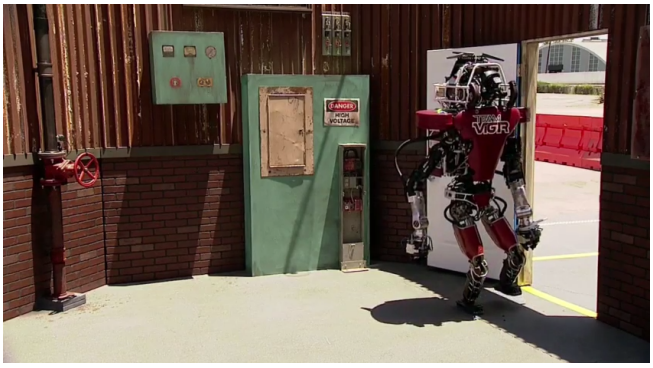


Fig. 1: Team ViGIR's Atlas humanoid robot on the first day of the DRC Finals. (Photo credit: DARPA)

However, writing a formal, logic-based specification is a non-trivial task that requires expert knowledge. In this paper, we automatically generate the logic-based specifications from a higher level, partial, multi-paradigm specifications: a description of the system's capabilities, the task's goals, and the task's initial conditions. Furthermore, most approaches in this field assume that the simple, low-level components that make up the high-level plan will work as expected, i.e., they never fail. In this paper, we take a first step towards lifting this assumption by formally accounting for the possibility of failure when executing the low-level components. We achieve this by generalizing the concepts of "activation" and "completion", which were introduced in [4] to deal with the time semantics of logic-based specifications. While there might be no way to recover from a failure, we can still achieve *graceful degradation*. That is, we want to specify the system's reaction to failure in the formal specification.

Furthermore, ours is an end-to-end approach that starts with an informal specification and results in an executable software implementation of a high-level plan. We first create a discrete abstraction of the problem and automatically construct a formal task specification in a fragment of Linear Temporal Logic (LTL). We then synthesize a *reactive* mission plan that is guaranteed to satisfy the formal specification. Finally, in an effort to bridge the gap between theoretical results and practice, we automatically generate the implementation of a state machine that instantiates the synthesized plan in software.

In this paper, we present and experimentally validate the proposed approach in the context of a Boston Dynamics Atlas humanoid robot running the software that Team ViGIR developed for the DRC (Fig. 1). However, the concepts apply to different systems. We have implemented and open-sourced

the proposed approach as a collection of Robot Operating System (ROS) packages [5], [6].

Related Work: Our work generalizes that in [4], which allows us to reason about multiple outcomes of an action, such as failure, in the formal specification. The approaches in [7] and [8] deal with the related problem of uncertainty in mobile robot motion. In terms of our approach to creating the formal specification, some other options would have been asking the user to (i) write the formal specification, e.g., LTL formulas, directly, (ii) write Structured English statements [9], or (iii) specify the task in natural language [10]. Each option comes with trade-offs and we chose one where the user input is essentially minimal. In terms of the mission planning step, we opted for GR(1), i.e., reactive LTL, synthesis [11] over other approaches. These included classical AI planners, such as STRIPS [12] and PDDL [13], optimization under LTL constraints [14], and, most notably, synthesis from co-safe LTL specifications, e.g., [15]. Our main reason for choosing GR(1) synthesis is the ability to specify reactivity w.r.t. a dynamic, and even adversarial (worst-case), environment (such as external events and component failures). Finally, [16] also presents an end-to-end approach (from formal specification to software generation), while the toolkit presented in [17] employs an executive that executes the abstract synthesized automaton. However, in both of works, the user has to write a Structured English [9] specification that exactly maps to LTL, a non-trivial task. In our work, the user input is a partial, informal specification.

The rest of this paper is organized as follows. In Section II, we introduce Atlas, Team ViGIR's approach to the DRC, and Linear Temporal Logic. In Section III, we state the problems that this paper addresses. Sections IV through VI present the proposed approach, while Section VII summarizes its ROS implementation. We present experimental demonstrations in Section VIII. Finally, we draw conclusions and propose future research directions in Section IX.

II. 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.

Atlas is equipped with a number of sensors, most notably a Carnegie Robotics Multisense SL[1] mounted as the head. For the DRC Finals, Atlas was equipped with two Robotiq 3-finger hands,[2] providing basic manipulation capabilities.

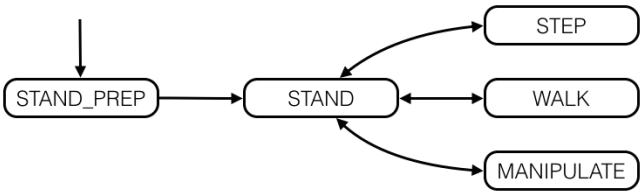


Fig. 2: Excerpt from the BDI control mode interface. Some changes between modes (depicted as arrows) are unidirectional while others are bidirectional. After Team ViGIR's initial checkout, Atlas is in the STAND_PREP control mode.

B. Team ViGIR's Approach to the DRC Finals

Team ViGIR has based its software on the Robot Operating System (ROS). In this section, we highlight some elements of the software's design that are relevant to high-level control and behavior synthesis. For a complete overview of Team ViGIR's approach, we refer the interested reader to [18]. 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).

In terms of manipulation, Team ViGIR employs the concept of object templates [19]. 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[3] (FlexBE) [1], [2], which is a major extension of the SMACH high-level executive [20].

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,[4] which encode the logic of execution as well as the flow of data. 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

[1] http://carnegierobotics.com/multisense-sl
[2] http://robotiq.com/products/industrial-robot-hand
[3] https://github.com/team-vigir/flexbe_behavior_engine
[4] https://github.com/team-vigir/vigir_behaviors

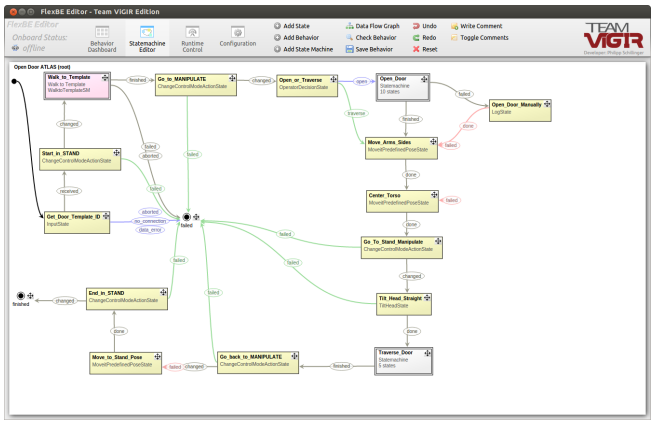


Fig. 3: A manually designed state machine for carrying out the DRC Finals' "Door" task. The initial state is indicated by the black arrow originating from the top left. The state machine has two outcomes, "finished" (bottom left) and "failed" (center). Yellow states are parametrized state implementations. 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.

to specific control modes, e.g., to STAND. State machines also have outcomes themselves, e.g. {`finished`, `failed`}.

Finally, both composition of state machines and supervision of their execution takes place in FlexBE's graphical user interface[5] (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 [11]. 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.

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

GR(1) synthesis involves setting up a two-player game between e and s [11]. 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.

III. Problem Statement

We are motivated by three main considerations. First, we want to formally specify the task to be carried out by our system in a way that also captures the possibility of failure and allows for the system's reaction to failure. In addition, we want to allow non-expert users to create such formal specifications. Finally, we want to automatically generate a mission plan that is verifiably-correct w.r.t. the formal specification, as well as its software implementation. These considerations give rise to the following problem statements.

Problem 1 (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 the execution and outcomes of its primitive capabilities (control mode transitions and actions). In addition, maintain a mapping, $\gamma : \mathcal{D} \rightarrow \mathcal{C}$, which grounds[6] the abstract symbols of the discrete abstraction to the primitive capabilities $\mathcal{C}$ of the system $\mathcal{S}$.

Problem 2 (Formal Task Specification): Given a task in terms of goals $\mathcal{G} = \{g_1, g_2, \ldots, g_n\}$, the task's initial conditions $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$, and the discrete abstraction $\mathcal{D}$ of the system $\mathcal{S}$ that is to carry out the task,[7] automatically generate a specification $\mathcal{T}_\mathcal{S}$ that encodes, in a formal language, the task being carried out by $\mathcal{S}$.

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

Problem 3 (Behavior Synthesis): 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.

IV. 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

[6] We refer the reader to [21] for a review of symbol grounding in robotics.

[7] 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.

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., $Prec(a) \in 2^{(\mathcal{A} \cup \mathcal{M})}$ (power set) and $a \notin Prec(a)$, $\forall a \in \mathcal{A}$.

B. Atomic Propositions

We adopt a paradigm that generalizes the one in [4]. 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,[8] 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 will abstract all positive outcomes as "completion" (c) and all negative outcomes as "failure" (f). That is, $Out(a) = Out(m) = \{c, f\}$, $\forall a \in \mathcal{A}$ and $\forall m \in \mathcal{M}$.

To conclude this section, we could 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 VI.

V. 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, unless a natural language based approach is used [10]. 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 user should only have to specify the goals without worrying about the internals of the robot and the software it is running. We can infer which actions 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 are either specified by the user or detected at runtime.

Thus, referring to Problem 2, we get the goals, $\mathcal{G}$, and initial conditions, $\mathcal{I}$, from the user. The discrete abstraction, $\mathcal{D}$, is system-specific and we assume that it has been created a priori from expert developers, according to Section IV. We can now automatically generate the task specification $\mathcal{T}_\mathcal{S}$ in (the GR(1) fragment [11] 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 (c.f. Section II-C). Conversely, we do not control the outcome of activation; the environment does. Therefore, the LTL formulas specifying the behavior of outcomes will be in φ_e.

1) Generic Formulas: We say that an activation proposition π_p, $p \in \{a, m\}$, is `True` when the corresponding primitive functionality is being activated and `False` when it is not being activated[9]. Therefore, the system safety requirement (3) dictates that all activation propositions $\pi_p \in \mathcal{Y}$ 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.

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

The environment safety assumption (4) dictates that the outcomes, π_p^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`.

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

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 [4].

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

2) Action-specific Formulas: The following formulas encode the connection between the activation and the possible outcomes of the robot'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. The disjunct $\neg \pi_a$ is added in order to prevent situations where the environment loses the game due to the system never activating the action.

[8] The distinction between action and control mode propositions is purely for the sake of clarity of notation. There is nothing special about either.

[9] Note that this is in contrast to [4], where π_p being `False` stands for the primitive functionality p being *deactivated*, e.g., turning a camera off.

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

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

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

where the superscript $c \in Out(p)$ 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 IV-A, Fig. 2) in LTL.

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

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

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

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.

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

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.

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

The environment liveness assumption (13) is the equivalent of the fairness condition (7) for control modes. A single formula suffices for mutually exclusive propositions [4].

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

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

C. 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 (14). All other outcome propositions corresponding to those actions and control modes, as well as all outcome propositions corresponding to any other actions and control modes, should be `False`.

$$\varphi_i^e = \bigwedge_{i \in \mathcal{I}} \Big(\pi_i^c \bigwedge_{o \in Out(i) \setminus \{c\}} \neg \pi_i^o \Big) \wedge \bigwedge_{j \notin \mathcal{I}} \bigwedge_{o \in Out(j)} \neg \pi_j^o \tag{14}$$

Activation propositions are `False` regardless of whether that action or control mode is in the initial conditions or not (15). The intuitive reasons is that if we want something to be an initial condition, then the resulting plan should not activate it at the beginning of execution.

$$\varphi_i^s = \bigwedge_{i \in \mathcal{I}} \neg \pi_i \wedge \bigwedge_{j \notin \mathcal{I}} \neg \pi_j \tag{15}$$

D. Specification of Task Goals

So far, we have handled the system-specific portion of $\mathcal{T}_{\mathcal{S}}$ and the initial conditions, $\mathcal{I}$. All that is left is the automatic generation of formulas from the user-specified task goals, $\mathcal{G}$. Motivated by the DRC tasks, we will present LTL formulas that encode the accomplishment of each goal once. However, LTL and its GR(1) fragment can naturally handle repeating tasks (e.g. patrolling). In the context of graceful degradation, we could even combine the two. For example, "accomplish the goals $\mathcal{G}$ infinitely often, but if anything fails, abort".

The system initial condition (16), safety requirements (17) and (18), and liveness requirement (19) are used to reason about the satisfaction of the system's goals, $g \in \mathcal{G}$, in a finite run (using the same LTL semantics as for infinite execution). In this paradigm, we say that the run itself has outcomes too. We denote them by $o \in Out(SM)$, since they will correspond to the outcomes of the synthesized plan, which will be a state machine (SM). Note that the propositions corresponding to the run's outcomes, π_{SM}^o, are system, not environment, propositions. We also introduce auxiliary system propositions, μ_g, which will serve as memory (c.f. [22]) of having accomplished each goal $g \in \mathcal{G}$.

$$\bigwedge_{g \in \mathcal{G}} \neg \mu_g \tag{16}$$

$$\bigwedge_{g \in \mathcal{G}} \square \big(\bigcirc \pi_g^c \vee \mu_g \Leftrightarrow \bigcirc \mu_g \big) \tag{17}$$

$$\square \Big(\pi_{SM}^c \Leftrightarrow \bigwedge_{g \in \mathcal{G}} \mu_g \Big) \tag{18a}$$

$$\square \Big(\pi_{SM}^f \Leftrightarrow \bigvee_{\pi \in \mathcal{Y}} \pi^f \Big) \tag{18b}$$

$$\bigwedge_{o \in Out(SM)} \square \Big(\pi_{SM}^o \Rightarrow \bigcirc \pi_{SM}^o \Big) \tag{18c}$$

$$\square\diamondsuit\Big(\bigvee_{o\in Out(SM)}\pi^o_{SM}\Big) \tag{19}$$

The formulas above can be interpreted as: "If nothing fails, then eventually accomplish each goal. Otherwise, abort".

Formula (17) 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 (17) with (20), whenever strict goal order is desired.

$$\bigwedge_{i=1}^{n}\square\left((\pi_{g_i}\wedge\bigcirc\pi^c_{g_i})\wedge\mu_{g_{i-1}}\vee\mu_{g_i}\Leftrightarrow\bigcirc\mu_{g_i}\right), \tag{20}$$

where $\mu_{g_0} \triangleq$ `True`. Formula (20) 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:

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

VI. High-level Behavior Synthesis

We tackle Problem 3 in two sequential steps. First, we automatically generate a correct-by-construction automaton from the formal specification $\mathcal{T}_S$ using GR(1) synthesis (see [11] and Section II-C). Specifically, we employ the synthesis algorithm in [23], which can handle a slightly larger fragment of LTL than GR(1). Namely, the one that includes $\bigcirc$ (next) operators in liveness formulas, such as in formulas (7) and (13). This algorithm was first used in [4].

Second, we use the mapping $\gamma : \mathcal{D} \to \mathcal{C}$ to instantiate the abstract automaton as a concrete software implementation, i.e., an executable state machine in the FlexBE framework introduced in Section II-B.

VII. ROS Implementation

We have implemented all aspects of our approach in `vigir_behavior_synthesis`,[10] a collection of Robot Operating System (ROS) Python packages. Figure 4 depicts these packages as well as the nominal workflow.

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 V is delegated to our "Reactive Specification Construction kit" (ReSpeC),[11] which is a Python framework with rudimentary ROS integration.

[10] https://github.com/team-vigir/vigir_behavior_synthesis

[11] https://github.com/team-vigir/ReSpeC

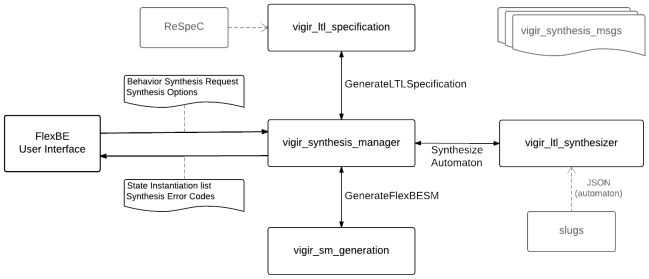


Fig. 4: Team ViGIR's "Behavior Synthesis" ROS packages and the nominal workflow (clockwise, starting from the left).

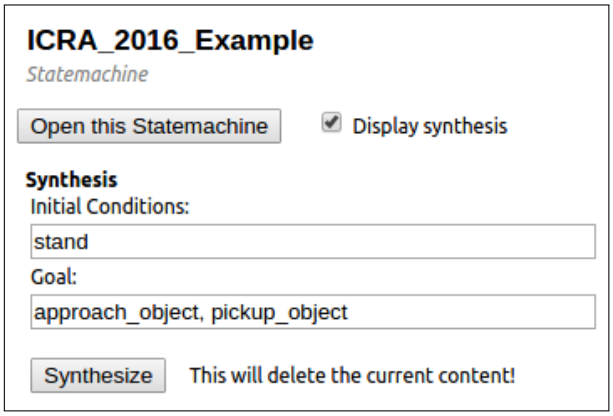


Fig. 5: Screenshot of the FlexBE Editor's synthesis menu.

The `vigir_ltl_synthesizer` package acts as a wrapper for external synthesis tools (currently, [23] is supported). Given the generated LTL specification, the `SynthesizeAutomaton` service returns a finite-state automaton that is guaranteed to satisfy it, if one exists. Finally, 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.

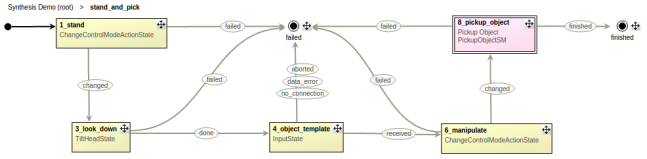
The following excerpt[12] is taken from the `StateInstantiation` list message, which is the end product of the `vigir_behavior_synthesis` workflow (Fig. 4). Specifically, this excerpt corresponds to the primitive functionality `object_template`, which appears in Fig. 6a.

```
state_path: /4_object_template
state_class: InputState
parameter_names: [request]
parameter_values: [InputState.SELECTED_OBJECT_ID]
outcomes: [no_connection, aborted, received, data_error]
transitions: [failed, failed, 6_manipulate, failed]
autonomy: [0, 0, 0, 0]
userdata_keys: [data]
userdata_remapping: [template_id]
```

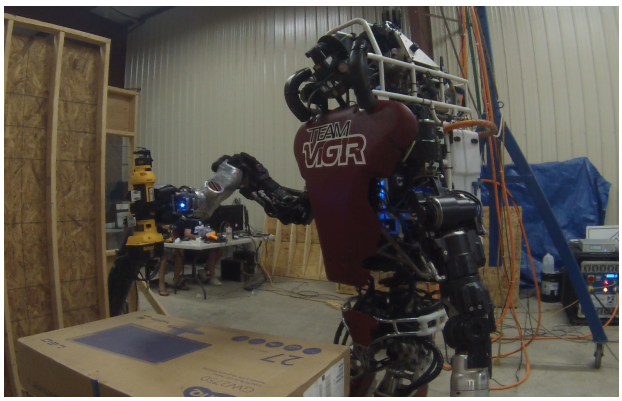
VIII. Experimental Validation

Team ViGIR did not employ high-level behavior synthesis during the DRC Finals. However, we later carried out experi-

[12] We have omitted some details for the sake of brevity and clarity of presentation. For example, most list elements are strings, e.g., `"template_id"`.

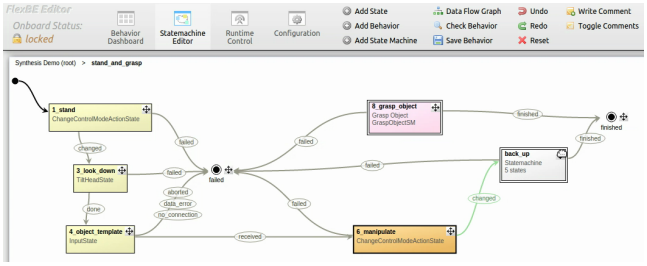


(a) The state machine above was synthesized for the task with $\mathcal{I} = \{\texttt{stand\_prep}\}$ and $\mathcal{G} = \{\texttt{look\_down}, \texttt{pickup\_object}\}$. The capability `object_template` requests an object template from the operator (Section II-B). It is a precondition of `pickup_object`.

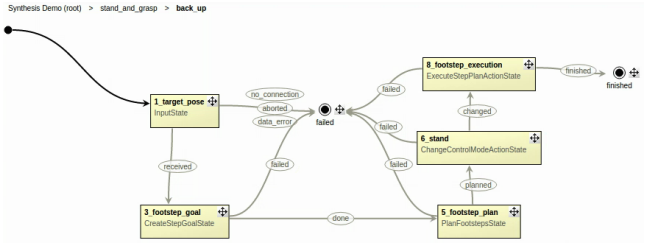


(b) Atlas finishing execution of state `8_pickup_object`.

Fig. 6: Snapshots from the demo in Section VIII-A.



(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 demo in Section VIII-B. In response to some unexpected event, the operator synthesized a state machine that has Atlas back away (7b).

mental demonstrations on Atlas in the lab. Due to a hardware issue, Atlas could not locomote. Thus, in addition to two experimental demonstrations, we present a simulation run carried out in Gazebo, using the same operator and onboard software. We summarize these demonstrations below. Please also refer to the accompanying video.

A. Behavior Development using Synthesis

In the first experimental demo, we show how a high-level behavior is specified and synthesized starting from scratch[13]. Once the state machine has been instantiated (Fig. 6a), it is ready for execution (Fig. 6b).

B. Online Modifications using Synthesis

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 (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 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.

IX. Conclusions and Future Work

In this paper, we presented an end-to-end approach to high-level mission planning. We combine an informal task specification provided by the user with a discrete abstraction of the robot and software system to automatically generate a formal specification in the GR(1) fragment of Linear Temporal Logic. We then synthesize a verifiably-correct reactive mission plan. Finally, we automatically generate a software implementation of the mission plan in the form of an executable state machine. We implemented our approach as a collection of Robot Operating System packages and experimentally validated it on a Boston Dynamics humanoid robot running the software that Team ViGIR developed for the DARPA Robotics Challenge.

It is important to note that there is a 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 the use of template formulas or hard-coded assumptions.)

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 does it mean to offer formal guarantees when the activation of robot capabilities can result in failure?" We plan on further exploring this research direction.

In addition, we are interested in automating another step of our approach, the construct of the discrete abstraction. Currently, an expert user had to construct it (once for each system). We believe that by formally specifying the capabil-

[13]The LTL specification and the synthesized automaton are available at: https://gist.github.com/spmaniato/c37fb12e874c73d986da

ities and requirements of individual system components, we will be able to automatically the discrete abstraction, which includes action preconditions, action outcomes, etc. Finally, we will be demonstrating our approach on a number of other, more accessible, robotic systems.

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, "Development of an Operator Centric Behavior Control Approach for a Humanoid Robot," Bachelor's thesis, Technische Universität Darmstadt, 2013.

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

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

[4] 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.

[5] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

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

[7] 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.

[8] I. A. Sucan and L. E. Kavraki, "Accounting for uncertainty in simultaneous task and motion planning using task motion multigraphs," in *IEEE International Conference on Robotics and Automation*, St. Paul, May 2012, pp. 4822–4828.

[9] 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.

[10] 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. [Online]. Available: http://dx.doi.org/10.1007/s10514-014-9418-8

[11] 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.

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

[13] 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.

[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] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, "Towards manipulation planning with temporal logic specifications," in *2015 IEEE Intl. Conf. Robotics and Automation (ICRA)*. Seattle, WA: IEEE, 26/05/2015 2015, pp. 346–352.

[16] 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, September 2015.

[17] 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.

[18] 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.

[19] 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 *Proc. IEEE-RAS Intl. Conf. Humanoid Robots*, Madrid, Spain, Nov 2014, pp. 979–986.

[20] 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.

[21] 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.

[22] V. Raman, B. Xu, and H. Kress-Gazit, "Avoiding forgetfulness: Structured English specifications for high-level robot control with implicit memory," in *IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems*, 2012.

[23] R. Ehlers, V. Raman, and C. Finucane. (2013–2015) Slugs GR(1) synthesizer. [Online]. Available: https://github.com/VerifiableRobotics/slugs