Guidelines for a Human-Robot Interaction Specification Language

David Porfirio,[1] Mark Roberts,[2] and Laura M. Hiatt[2]

Abstract— **Designing novel *application development environments* (ADEs) is a growing area of systems research within the human-robot interaction (HRI) community. This research involves the design of a novel system, the ADE, to afford end users and application designers the ability to develop robot applications. Researchers then usually validate their ADEs in the form of user studies or a series of case studies. In this paper, we highlight a problem with the typical approach to conducting ADE research within HRI—there is currently little standardization in how these systems are designed, developed, and validated, leading to difficulty in sharing resources between different research groups and the inability to compare similar ADEs to each other. We argue that a standardized formal representation embedded within an *Interaction Specification Language* (ISL) can lead to more streamlined development and validation of ADEs for HRI. Furthermore, we discuss several desired characteristics that an ISL should embody.**

I. Introduction

The human-robot interaction (HRI) community has produced significant advances in designing *application development environments* (ADEs) that facilitate the construction of social, service, or collaborative robot *programs*—static specifications of a robot's actions and decisions that it should make when faced with internal and external stimuli. ADEs usually include a number of predefined, but general, descriptive primitives for a robot platform (*e.g.,* move, carry, lift, hold), as well as operational models (*e.g.,* motion plans, behaviors) that can be executed to complete or carry out those descriptive models. Crucially, the use of an ADE to specialize a robotic platform to a particular application is usually done by domain experts or robot end users. Any low-level programming (*e.g.,* functions for sensing and actuation) that is typically done by robotics engineers is integrated within the ADE and robotic platform, and is not usually exposed to the ADE end user. Researchers in academia and industry have designed a wide range of ADEs, often either for research or market use. Notable examples include *CoSTAR* [1] and *Moveit! Studio* [2] for developing collaborative robots, *iCustomPrograms* [3] and *Vipo* [4] for service robots, and *Choregraphe* [5] and the trigger-action programming *Tailoring Environment* for social robots [6].

Although ADEs can be created for a variety of purposes, our focus is on those created for research, not (at least, yet) for the market. ADE research goals are often to improve HRI application development for technical non-experts and robot end users by way of investigating novel interfaces and development paradigms. To pursue these goals, a researcher (or a team of software engineers working under the direction of the researcher) must implement their design into a full-fledged system—the ADE itself. The ADE must then be *validated*. Validation of an ADE often occurs through a test suite, demonstrations, or user testing, *i.e.,* asking people to use the platform to program a robot or interact with a robot executing programs created by the platform. The purpose of validation is to show that the ADE achieves the research goals set at the beginning of the project.

Unfortunately, relatively little standardization exists in how researchers perform ADE design and validation, leading to a diverse, yet inconsistent array of ADEs with a limited ability to compare between them. One crucial component of ADE design that suffers from a lack of standardization is the underlying *formal representation* of robot programs, which defines the semantics of human-robot interactions (*e.g.,* robot actions, human behaviors, external and internal events, working memory, world state, etc.). Without a consistent underlying formal representation (which we sometimes refer to as simply *representation*), it is difficult to compare ADE functionality, undergo consistent performance evaluations to situate the technical capabilities of one ADE with respect to others (*i.e.,* ADE^1 guarantees that loops terminate, whereas ADE^2 makes no such guarantee), and benchmark the quality of programs produced by potential end users of different platforms (*e.g.,* evaluating whether users of ADE^1 produced programs of significantly higher quality than users of ADE^2).

This lack of standardization also presents a missed opportunity to improve the efficiency of ADE research. Rather than being able to use and adapt off-the-shelf code to streamline ADE development, researchers often must develop their own *ad hoc* representations and runtime environments from the ground up. Furthermore, researchers are unable to exploit common ADE performance benchmarks, such as test suites, that would streamline ADE validation.

To address these issues, we advocate for the creation of a standardized formal representation for HRI programs embedded within a human-robot *Interaction Specification Language (ISL)*. Figure 1 illustrates how an ISL could be situated within a basic set of stakeholders of ADE design: the ADE Researcher (center), their Industrial Partner (left), and the ADE End User (right). We believe that having a shared representation between researchers and their industrial partners would increase resource sharing, thus streamlining the ADE research process and improving ADE validation.

This research was supported by the Office of Naval Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the US Navy.

[1]David Porfirio is an NRC Postdoctoral Research Associate at the U.S. Naval Research Laboratory, Washington, D.C., USA. david.porfirio.ctr@nrl.navy.mil

[2]Mark Roberts and Laura M. Hiatt are with the U.S. Naval Research Laboratory, Washington, D.C., USA. {first}.{last}@nrl.navy.mil

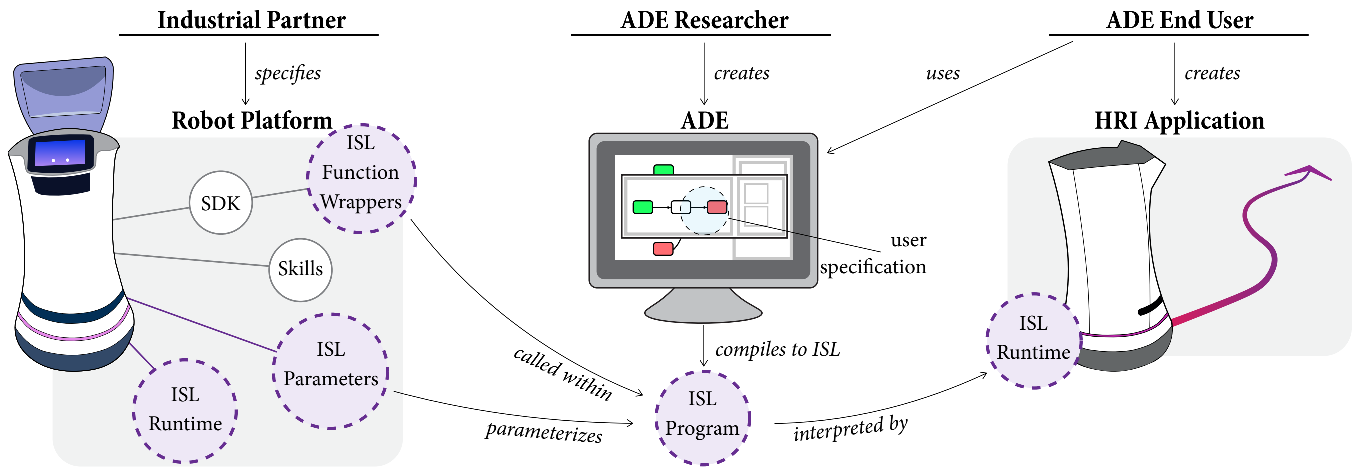

Fig. 1. An illustration of how the ISL can fit within the HRI application development process.

In what follows, we propose six guidelines for the creation of an ISL, concretizing each guideline with a simple, speculative language that we created ourselves. We introduce our illustrative, example language below within a "notional grammar." In this paper, our notional grammar is intended purely for demonstration purposes and to generate discussion about the shared responsibility of academia and industry to bring about an ISL to fruition.

II. Notional Grammar

For a standardized ISL, we envision a singular, common domain-specific language or a family of languages for human-robot interaction that is akin to the *Planning Domain Definition Language* (PDDL) [7] for automated planning,[1] *Game Description Language* (GDL) [9] for game theory, or agent-oriented languages for *Belief-Desire-Intention* agents (*e.g.,* [10], [11]). Although certain categories of languages and formal representations dominate within human-robot interaction ADE research, no single language is yet considered standard. We may either adopt an existing language as the basis of our ISL or develop a new language from scratch.

To help illustrate the considerations for developing a future ISL, we have created a simple notional ISL grammar that fits within the state or flow-based class of representation popular with many existing ADEs (*e.g.,* [4], [5], [12]–[15]). Our notional grammar contains nodes n that describe a robot's current state, actions, and/or goals; events e, or any internal or external events that the robot can sense; and guards g, or conditionals on the state of the world. We refer to [16] for a distinction between guards (called "states" in [16]) and events. An expression subset of the grammar describes a start node and the transition relations between nodes n:

$$\begin{array}{rlcl}
\text{Initialization} & \mathit{begin} & := & \textbf{begin}\ n \\
\text{Transition} & \mathit{trel} & := & [e]\ n\ (g)\ \rightarrow\ n \\
\text{Event} & e & := & \mathit{id}\ \mid \texttt{nil} \\
\text{Guard} & g & := & m\ \phi\ \mathit{value} \mid g\ \psi\ g\ \mid \texttt{nil}
\end{array}$$

A transition `[e] n1 (g) -> n2` can be interpreted as, "When event e occurs, if guard g is satisfied then node *n1* will transition to node *n2*." An event is denoted by a string identifier. For the guard, m is a variable in the robot's memory, ϕ is a relational operator that compares m to some *value*, and ψ is a logical operator. Another expression subset of the grammar defines memory, nodes, and expressions:

$$\begin{array}{rlcl}
\text{Nodes} & \mathit{ndec} & := & \textbf{let}\ n\ =\ \mathit{expr} \\
\text{Memory} & \mathit{mdec} & := & \textbf{var}\ m \mid \textbf{var}\ m\ \leftarrow\ \mathit{value} \\
\text{Expression} & \mathit{expr} & := & \texttt{func}(\mathit{parameters}) \\
 & & & \mid m\ \leftarrow\ \texttt{func}(\mathit{parameters}) \\
 & & & \mid m\ \leftarrow\ \mathit{value} \\
 & & & \mid \mathit{expr}\textbf{,}\ \mathit{expr}
\end{array}$$

Here, $\leftarrow$ refers to memory assignment and vertical bars are metasyntax. Commas (,) denote concurrency. The **`func`** token refers to a function over the robot's capabilities.

To illustrate, consider the code snippet in Figure 2 (also visualized in Figure 6c), in which a delivery robot transfers an item from its tray to a hotel guest. Line 1 (L1) declares a variable to store `rating`. L2-6 label nodes n1-5 with robot actions. In L4, we can assume that **`queryRating`** is defined so that the robot visually prompts the person to rate its performance and returns a value on a scale from one to five. L7 begins the program at n1, in which the robot opens its tray. L8-12 define the transition relations of the program—after hearing the *`deliveryReceived`* event, the robot will

```
 1  var rating
 2  let n1 = openTray
 3  let n2 = closeTray
 4  let n3 = rating <- queryRating
 5  let n4 = happyDance
 6  let n5 = goodbye
 7  begin n1
 8  [deliveryReceived] n1 -> n2
 9  [] n2 -> n3
10  [] n3 (rating>=4) -> n4
11  [] n4 -> n5
12  [] n3 (rating<4 or time>20) -> n5
```

Fig. 2. Code snippet for making a delivery.

[1]There are a variety of other languages for automated planning (*e.g.,* ANML [8]) that we plan to explore as possible directions for the ISL.

close its tray (L8) and ask the person for a rating (L9). If it receives a high rating, the robot will perform a **`happyDance`** (L10) and say **`goodbye`** (L11). Otherwise, if a low rating is received or more than 20 seconds have passed, it will say **`goodbye`** without performing a dance (L12).

Like other candidate representations for an ISL, our notional grammar has both benefits and drawbacks, which serve as a basis of discussion in sections §III and §IV.

III. Proposed ISL Guidelines

Using our notional grammar for illustration, we describe our guidelines for creating an ISL. Each guideline is guided by our goal of streamlining ADE design and facilitating better and more consistent validations. The first three guidelines refer to the ability of the ISL to (1) *support existing formal representations*, (2) *support different levels of expressiveness*, and (3) *support predefined and reusable skills*. The next three properties pertain to the ISL being (4) *verifiable at design time*, (5) *consistent at runtime*, and (6) both *application and domain agnostic*. Each guideline is accompanied by an example code snippet within our notional grammar for illustration, followed by implications for academic and industrial practitioners.

A. Support Existing Formal Representations

Though no standardized formal representation exists across modern ADEs, there are many current, well-known representations that individual ADEs use for encoding programs. For example, users of *CoSTAR* create programs as behavior trees [1]; users of *RoVer* create programs as transition systems [14]; and users of the *situated live programming* platform create event-driven trigger-action programs [17].

The wide variety of existing formal representations underlying different ADEs attests to the unique benefits and drawbacks that each representation affords. Choosing an ideal representation is, in fact, a key component of researching novel ADEs. The chosen representation of an ADE affects user experience (*e.g.,* trigger-action programming is more accessible to programming novices) and the ease of adopting state-of-the-art software engineering techniques such as verification, synthesis, and repair (*e.g.,* transition systems can often be used as direct input to program verification techniques [18]). The underlying representation of an ADE can further restrict the functionality of the robot (*e.g.,* trigger-action programming limits the ability to designate a sequence of behaviors for the robot to perform). Given the meticulousness required of choosing a representation for a specific application domain or use case, we believe that enforcing a uniform formal representation without regard to individual circumstances would be careless. Rather, the role of the ISL should therefore be to *support* these existing representations by serving as a common, underlying representation. In essence, ADE researchers can choose any front-end representation to expose to their end users or back-end representation upon which to perform computation, but any programs generated by the ADE should ultimately be compilable to the ISL.

Case Study. Many existing languages and representations for human-robot interactions are equivalent under Chomsky's hierarchy, so converting to or from an ISL will be straightforward. Consider the block-based greeter program crafted using the *iCustomPrograms* ADE (see [3], Figure 1b). In this program, the robot detects people, chooses a random person, and then moves toward that person. If interrupted in its approach, the program begins a *while* loop of requesting space and subsequently resuming navigation. Upon reaching its destination, the robot will greet the person. To illustrate how an ISL can operationalize a common, underlying representation to which other representations can convert, we have approximated [3]'s greeter program within our notional grammar in Figure 3.

```
 1  var person
 2  var currLoc
 3  let n1 = findPeople
 4  let n2 = person <- chooseRandomPerson
 5  let n3 = currLoc <- moveTo: person
 6  let n4 = display: "Please allow me to pass."
 7  let n5 = greet
 8  begin n1
 9  [] n1 -> n2
10  [] n2 -> n3
11  [interrupted] n3 -> n4
12  [] n4 -> n3
13  [] n3 (currLoc==person) -> n5
```

Fig. 3. Greeter program from [3] converted to our notional grammar.

L1-2 declare variables that store the chosen person to approach and keep track of the robot's location. L3-7 label nodes n1-5 with ISL expressions, with L4-5 assigning values to `person` and `currLoc`. L8 sets the initial state of the program to n1. Finally, L9-13 describe the node transitions. Note that node n3 must transition to n4 if the robot observes the *`interrupted`* event, while n3 can only transition to n5 if the robot observes that it is close to the person.

Implications. While we envision research groups being able to employ their own favorite "surface representations," an underlying representation, the ISL, must still be decided on. A logical solution would be to choose an already popular representation to minimize the compilation effort of existing surface representations to the ISL. The state-based representation is popular in ADE design and is the basis for our notional grammar, but this representation has limitations. For example, state-based representations become unintuitive as complexity scales and require adaptation to incorporate certain control structures like for-loops. Alternatively, block-based programming is an emerging favorite among ADE researchers and industry partners but can be more difficult to analyze due to the underlying program flow being less explicit. Choosing a common underlying representation will necessarily involve additional discussion amongst all stakeholders of ADE design.

B. Support Different Levels of Expressiveness

Existing ADEs support different levels of expressiveness, *i.e.,* the level of detail at which programs can be specified. [12] describes an architecture that includes a high-level *application* layer and lower-level *behavior* and *information processing* layers. The application layer enables users with lower amounts of technical expertise to specify the flow of a human-robot interaction and the decisions that the robot may make (useful for technical non-experts), while the lower layers enable users with higher technical expertise to specify individual robot behaviors. Other ADEs extend behavioral-level expressiveness to technical non-experts, such as through *Choregraphe*'s keyframing functionality [5] and through *Puppet Master*'s use of programming by demonstration [19].

It seems clear that an ISL should support hierarchical constructions with functionality at different levels of abstraction. At the high level, the ISL should support functionality similar to existing approaches for specifying interaction flow, such as the encapsulation of robot behaviors within discrete actions and the symbolic representation of external phenomena (*e.g.,* human behavior and signals from Internet-of-Things (IoT) devices) as both triggers and world state. Furthermore, the ISL should be modular such that individual agents (*e.g.,* the robot together with any human and IoT devices in the vicinity) can be specified separately with shared resources.

At the low level, an ISL should support functionality similar to existing robot programming frameworks and software development kits (SDKs). Examples include the robot operating system (ROS), which facilitates asynchonicity and parallelism [20], and NaoQi, which enables the manipulation of precise joint angles, continuous-time behaviors, and raw sensor input [5]. The ability to encode low-level functionality into a formal representation is crucial to the existence of development platforms that support keyframing (*e.g.,* [5]) and programming-by-demonstration (*e.g.,* [21]).

It is clear that our vision of the ISL involves support for multiple components at each level of the hierarchically modular design, including such details as continuous time and precise joint angles, to name some examples. However, a singular language that captures all of these details will be cumbersome to create, difficult for ADE researchers and their industry partners to adopt, and inflexible to trends in the field. Therefore, we propose that an ISL not encompass a single all-purpose formal representation, but instead an extendable and parameterizable family of representations, similar to the different versions and extensions of PDDL.

Case Study. To enable different levels of program expression, we require an extension of our notional grammar that supports modularity and hierarchy. Using this extension, we expand our hotel robot example to a hierarchical specification with four separate modules, visualized in its entirety within Figure 6. In addition to including the code snippet from Figure 2, the expanded example incorporates higher-level task flow that guides the robot in receiving orders and moving to drop-off points, low-level actuation, and battery management. If the robot's battery becomes too low, the robot must halt its delivery, discard undelivered items, and charge its battery.

Figure 4 defines high-level task flow. L1 instantiates global state—whether the robot is currently charging its battery. L2 begins a *main* module. Within *main*, L3 instantiates `trayOccupied` to track whether the robot's tray is full, and L4 declares `room` to store the robot's current delivery target. L5-9 assign expressions to nodes n1-5. Nodes n2, n4, and n5 are assigned multiple expressions each. In n2, the robot receives its order from the front desk and designates a `room`

```
 1  var isCharging <- false
 2  module main
 3    var trayOccupied <- false
 4    var room
 5    let n1 = moveTo: charging
 6    let n2 = room <- receiveOrder,
               trayOccupied <- true
 7    let n3 = moveTo: room
 8    let n4 = completeDelivery: room,
               trayOccupied <- false
 9    let n5 = cancelDelivery,
               trayOccupied <- false
10    begin n1
11    [] n1 (isCharging==false) -> n2
12    [] n2 (isCharging==false) -> n3
13    [] n3 (isCharging==false) -> n4
14    [batteryLow] n1, n2, n3 -> n5
15    [] n4, n5 -> n1
16  endmodule
```

Fig. 4. Main module to control a delivery robot.

```
 1  module battery
 2    var level
 3    let n1 = level <- checkBattery
               isCharging <- false
 4    let n2 = level <- chargeBattery,
               isCharging <- true
 5    begin n1
 6    [] n1 (level>=10) -> n1
 7    [batteryLow] n1 (level<10) -> n2
 8    [] n2 (level<100) -> n2
 9    [] n2 (level==100) -> n1
10  endmodule
```

Fig. 5. Battery module that runs parallel to the main module.

for the order, in addition to setting `trayOccupied` to true. In n4, the robot delivers the order and sets `trayOccupied` to false. Alternatively, the robot can cancel its delivery (n5) and set `trayOccupied` to false. L10 starts the program at n1. Finally, L11-15 describe the transition relations, with L14 listening for a *`batteryLow`* event that will cause the robot to cancel an in-progress delivery (n5). L16 ends *main*.

Battery usage is monitored and handled within a separate and concurrent module (Figure 5). L1 begins the *battery* module. L2 declares a variable to store the robot's battery level. L3-4 assign expressions to nodes n1-2, which update `level` by checking or charging the battery. L5 starts the program at node n1, and L6-9 describe the transitions. The robot continually monitors its battery while `level` is above or equal to 10% (L6). When `level` is below 10%, the robot will enter its charging state (L7), within which it will stay until its battery level is 100% (L8). When the battery level is 100%, the robot will stop charging (L9). L14 ends *battery*.

The *main* and *battery* modules run concurrently to each other and synchronize with the *`batteryLow`* event. Synchronization forces the transition on L14 of *main* to occur iff (if and only if) the transition on L7 of the *battery* module occurs. Note that if the battery level gets low when the robot is already at the target room (node n4, module *main*), the delivery must finish before *main* and *battery* can synchronize on the *`batteryLow`* event. Synchronization of concurrent transition systems is known as a *handshake* [18].

For hierarchy within the ISL, we can imagine defining robot functions as low-level ISL modules. For example, the

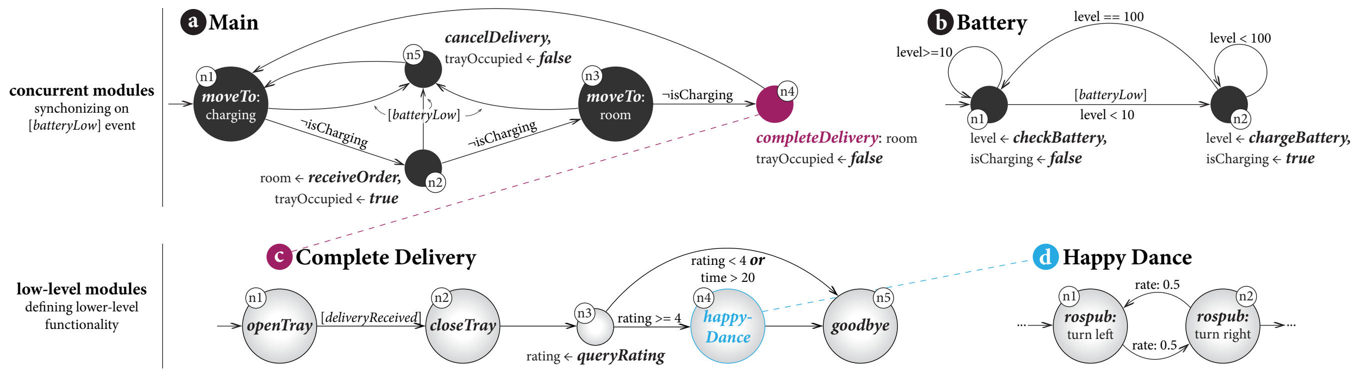

Fig. 6. An illustration of the hierarchical delivery specification, referencing code snippets in §II and §III-B. Filled circles (top) are nodes in the high-level, concurrent *main* and *battery* modules. Unfilled circles (bottom) are nodes in the *completeDelivery* and *happyDance* modules, which define lower-level functionality through ISL logic. Transitions are labeled with events (in brackets) and guards. *Main* and *battery* synchronize on the [*batteryLow*] event.

`completeDelivery` action within the *main* module may be defined by another *completeDelivery* module encompassing the code from Figure 2. Within *completeDelivery*, we can extend our notional grammar even further to control low-level actuation. As an example, to define the robot's **`happyDance`** action in Figure 2, perhaps a *happyDance* module contains nodes that are assigned ROS publishers that cause the robot to wiggle back and forth enthusiastically. The grammar must be further extended to set the rate at which the transitions between n1 and n2 occur (once every 0.5 seconds):

```
let n1 = rospub: "cmd_vel..." # turn left
let n2 = rospub: "cmd_vel..." # turn right
...
[] n1 -> 0.5 n2
[] n2 -> 0.5 n1
```

Implications. While our notional grammar has been shown to be hierarchically expressive, ADE researchers must decide which levels of the hierarchy are actually needed within the ISL. Low-level ISL modules such as *happyDance* may be unnecessary as these skills may be provided pre-trained or pre-engineered by an industrial partner. Researchers should not have to reimplement these skills within the ISL. The ability of the ISL to reuse existing, non-ISL skills is discussed in the next section.

C. *Support Predefined and Reusable Skills*

The purpose of ADEs in HRI is to facilitate task scripting, usually for a set of specific use cases, that guide a robot on how to perform a task or engage in a social interaction. However, not all components of an HRI platform need or should be formally specified in the ISL syntax.

The ISL must also support the integration of professionally engineered black-box robot skills, *e.g.,* visual servoing and autonomous navigation and localization. Skills should be able to be defined by ADE end users as well, such as if a user wishes to keyframe a waving behavior on a social robot. The ISL should additionally support non-hand crafted skills, *e.g.,* skills that are trained via machine learning. Learned, engineered, and user-defined black-box skills can then be treated as reusable functions available for parameterization and instantiation at different levels within the ISL's expressiveness hierarchy. At the low level, a skill may consist of an individual robot behavior or cue. At higher levels, a skill may consist of a learned interaction flow or high-level program structure, such as a conversational module.

The ISL should additionally recognize the ability of robots to make autonomous decisions via AI planning. It should therefore support the ability of ADE researchers and end users to label robot actions and skills with preconditions and positive and negative effects (postconditions).

Case Study. The case study presented in §III-B, *Support Different Levels of Expressiveness*, contains many examples of incorporating predefined reusable learned or engineered skills into ISL programs. As an example, within the robot's **`moveTo`** functionality (L5 and L7 of the *main* module), an ADE end user need not reason about the exact logic necessary to make the robot move to its destination. Instead, the user can leverage the robot's autonomous navigation and localization capabilities. Similarly, the **`checkBattery`** function (L3, module `battery`) likely corresponds to an engineered, proprietary, and closed-sourced function within the robot's SDK that is linked to the ISL.

Robot actions may also correspond to learned skills. Rather than writing an ISL module to define the robot's **`happyDance`** behavior, ADEs can leverage repeated interactions between the robot and hotel patrons to train this skill. Similar training can be performed with the **`openTray`** and **`closeTray`** behaviors within the *completeDelivery* module—the robot can leverage the ratings it obtains (**`queryRating`**) to learn an optimal policy for (a) notifying patrons of its arrival, (b) opening its tray, (c) waiting for patrons to remove items from its tray, and (d) closing its tray. The learned skills can then be leveraged and reused within the ISL logic.

Finally, as an example of how the ISL can integrate with an AI planner, consider the **`receiveOrder`** action (L6, *main* module). A minimalist implementation of **`receiveOrder`** may assume that the robot is already positioned near the front desk (where the order is ready to be picked up) and has opened its tray to receive the order. However, neither moving to the front desk nor opening its tray is included in the *main* module. Therefore, the minimalist implementation of **`receiveOrder`** must be accompanied by a task planner at runtime to achieve the preconditions of **`receiveOrder`**. Enumerating the pre and postconditions of **`receiveOrder`** is possible by defining the action within STRIPS-like syntax:

```
action receiveOrder()
          preconds:  at_location=front_desk
                     tray_open=true
         postconds:  tray_open=false
```

Implications. Numerous industrial constraints may inform the integration of predefined and reusable skills into the ISL. One key constraint is to ensure the protection of intellectual property. Another is to provide entry points in the ISL for each stakeholder involved in the development of a robot platform or a proprietary ADE. The role of management, engineers, designers, and external collaborators (*e.g.,* third-party developers) involved in the development of intellectual property should be a deciding factor for how each party would benefit from developing at the ISL level or at the level of defining reusable robot skills, and to what degree a robot's program logic should be allocated to these skills versus being defined within ISL. ADE researchers must converse with their industrial partners to understand these constraints.

D. Verifiable at Design Time

Programs produced by development platforms should be easily verifiable, *i.e.,* the logic should be easily shown to adhere to a set of correctness properties. An ISL must therefore embed an easily verifiable representation (*e.g.,* a transition system verifiable via model checking). Properties of interest must also be able to be represented within standard property-specification logic, such as linear temporal logic (LTL) for discrete systems, signal temporal logic (STL) for cyber-physical systems, or probabilistic linear temporal logic (PLTL) for probabilistic systems. Although the complex, expressive form that the ISL will likely assume is difficult to verify in theory, the modularity and hierarchical construction of the ISL (see §III-B) will greatly further its verifiability.

Case Study. Consider the case study from §III-B, *Support Different Levels of Expressiveness*. By modeling the case study within a model checker, numerous interesting properties can be checked. Consider a simple rule in LTL that the program should not deadlock: `G[F main=n1]`.

The $\mathbf{G}\phi$ and $\mathbf{F}\phi$ operators indicate that in all paths through a program, ϕ will hold **G**lobally throughout the program or at some point in the **F**uture. The above property therefore translates to, "From any node in the program (globally), all paths will lead back to the starting node in module *main* at some point in the future." We may also wish to ensure that the robot will never charge its battery while its tray is full: `!F[isCharging & trayOccupied]`. This property translates to, "There is no future state in the program in which the robot charges its battery while its tray is full." Both properties can be directly provided as input to an off-the-shelf model checker, such as PRISM [22]. Given the similarity of our notional grammar to the PRISM language, translating the delivery case study to a verifiable form is straightforward. Running the model checker leads to the conclusion that both properties are satisfied.

Implications. The ISL must be easily verifiable. In the above case study, our notional grammar is shown to be verifiable through its straightforward translation to a model checker, PRISM. It is then the responsibility of the HRI community, both academia and industry alike, to leverage the verifiable ISL to standardize and streamline ADE validation. Industrial partners should ship platform-specific ISL modules with properties that ensure safe operation of their robot or that their runtime environments are guaranteed to satisfy. ADE researchers in both academia and industry should additionally co-construct a repository of useful properties that can be used as quality metrics for ADE validation. This repository can be expanded as ADE research progresses.

E. Consistent at Runtime

Even if they satisfy a set of correctness properties, ISL programs on different robot platforms may be inconsistent at runtime due to platform and contextual variability. Even if using the same ADE, different research groups must sometimes construct *ad hoc* runtime environments to execute ISL programs. For example, if an ISL program (perhaps naïvely) assumes that a path exists to the robot's destination yet no path exists at runtime, then one runtime environment may cause the robot to wait forever while a different runtime environment equipped with a planner may replan a different sequence of actions for the robot to achieve the same effect.

The ISL should anticipate and enumerate runtime contingencies that are unspecified or underspecified by ADE end users. When an ADE is released to the research community, research teams may also provide a "contingency sheet" that specifies the default fail-safe protocols that an ISL-programmed robot will exhibit in the event of underspecified or unforeseen phenomena.

Case Study. With ROS, robot application developers can set runtime parameters to enable consistent performance across different platforms, such as the base velocity and acceleration of a mobile robot. For consistent execution of ISL programs, additional standardization is similarly required at the meta-level—outside of the program logic. Consider the case study from §III-A, *Support Existing Formal Representations*. Despite its simplicity, there is ample opportunity for platform or context-specific runtime contingencies to cause different robots to deviate in different ways from the expected program flow. Perhaps the ambient lighting inhibits the robot's ability to detect people, which may cause one robot platform to hang indefinitely, while another might replan its task or terminate the program altogether. As another example, if someone attempts to interrupt or interact with the robot as it attempts to execute an ISL program, different runtime environments may cause the robot to either pause its execution to handle the interruption or continue without acknowledgment. Each of these examples affects the flow of an ISL program.

With a *contingency sheet*, the ISL runtime environment can be controlled and enforce consistent program flow on different platforms. An example contingency sheet is below:

```
onDeadlock: requestAssistance
timeout: 20
onInterrupt: ignore
onTerminate: moveTo charging
```

Within this contingency sheet, if the robot experiences a deadlock, it will issue a **requestAssistance** action to notify a human of the issue, receive assistance, and resume execution once the issue has been resolved. To prevent the robot from waiting too long for an event to occur, a default timeout can be set for when the robot ceases listening for the event. If someone attempts to interrupt the robot, the robot will ignore them. Lastly, the robot will handle task termination by moving back to its charging station.

Implications. The enumeration of runtime contingencies can provide ADE researchers with greater control over their ISL runtime environments and help ensure that the same ISL program behaves similarly on different robot platforms. A weakness of our notional grammar, and of ADE development in general, is that these contingencies cannot necessarily be anticipated nor handled within program logic. Rather, these contingencies often arise at the platform level.

Therefore, we advocate that the adoption of an ISL encourages industry partners—the designers and developers of robot platforms—to assist in the standardization of the ISL runtime environment. Industrial entities should maintain and incrementally expand a shared repository of contingency parameters. When shipping robots to customers, industrial partners can provide a parameterized set of contingencies from within this repository. Going a step further, industry partners can also ship standard runtime environments with their robots that interpret ISL specifications natively, furthering the goal of consistent, cross-platform robot deployment. ADE researchers can provide support to their industrial partners by reporting when new, unforeseen contingencies arise, so that these contingencies may be added to the shared repository.

If not careful, however, contingency sheets may weaken the standalone expressiveness of the ISL itself. Contingency sheets must therefore exist solely as sets of overridable, default runtime parameters absent of program flow or logic. Additionally, the ISL should clearly differentiate runtime platform characteristics (*i.e.,* those within a contingency sheet) from core program characteristics (*i.e.,* those within the ISL).

F. Application and Domain Agnostic

The ISL should be nonspecific in its support for different robot form factors, application areas (*i.e.,* social, service, and collaborative robotics) and domains (*e.g.,* healthcare, manufacturing, education, etc.) within HRI. At a minimum, the ISL itself should therefore not contain any platform or domain-specific tokens. Instead, platform and domain specificity can be achieved by supporting SDK integration. ISL modules should therefore be constructed to wrap SDK functionality.

SDK integration provides an additional opportunity for industry to support the standardization of ADE research. Consider the multitude of robot platforms and their SDKs that support volume control. Industry partners can wrap SDK functionality within a standardized ISL module with a standardized module name and parameters, *e.g.,* `setVolume: level`. ADE researchers can then use the same ISL syntax to interface with any platform that supports volume control.

Case Study. Consider two separate social robot platforms with SDK support for changing robot volume. Suppose that the first platform provides Android support with the following function header: `void setVolume(int level)`. The second platform provides Python support for a similar function but with a different header: `set_volume(level)`. Assuming that the logic of each function is equivalent, different industrial entities may then agree on a standardized volume-control function header, `setVolume: level`. The following ISL code is then easily translatable to either robot:

```
1 import setVolume from android_robot
2 let n1 = setVolume: 80
```

L1 imports wrapped functionality from the Android robot, and L2 uses this functionality by setting the robot's volume. Changing from the Android to the Python robot requires modifying only the last part of the import statement from `android_robot` to `python_robot`.

Implications. Import statements help the notional grammar to be platform agnostic, but full domain and platform non-specificity will require substantial coordination between industrial partners. Cross-platform code must have nearly identical functionality for ISL programs to behave consistently on different platforms. To ensure consistency, industrial partners should decide on a common set of functions that possess similar inputs and outputs, *e.g.,* volume control. For robots with inconsistent functionality, ADE researchers can bear the responsibility of creating wrapper functions containing a combination of the platform SDK and ISL logic.

IV. Discussion

Our vision of a standardized human-robot interaction specification language, or ISL, is motivated by the need for more streamlined ADE design and validation. In what follows, we elaborate on the benefits that an ISL might have within the academic and industrial HRI communities and offer a vision of future work.

a) Potential ISL Benefits: We expect that an ISL would streamline the work of ADE researchers, who could more easily reuse existing ISL functionality. For example, after constructing and linking a novel ADE to the ISL, researchers can leverage a multitude of existing robot platforms with standardized ISL runtime environments. Furthermore, an ISL that follows our proposed guidelines would avoid the need for researchers to adopt unfamiliar formal representations. Instead, the ISL must support existing formal representations and existing levels of expressiveness to which researchers have already become accustomed.

We expect that an ISL would also ease the burden of validation, since the verifiable and predictable characteristics of the ISL would enable the creation and sharing of correctness properties (*e.g.,* in LTL) and runtime parameters. The creation and sharing of correctness properties and runtime parameters would additionally facilitate a better comparison between separate ADEs. A standard set of properties and parameters can be used as benchmarks within the HRI ADE research community (*e.g.,* under a specific set of runtime parameters, ADE^1 guarantees that its programs satisfy a greater number of properties than ADE^2). Furthermore, a standard program representation enables researchers designing an ADE to create and share programs with other researchers designing other ADEs. In our vision, research communities can use each other's programs as test cases that prove that their ADE can produce programs equivalent to existing ADEs.

Lastly, an ISL would help academic and industrial researchers bridge their work across the intellectual property boundary. A proprietary development platform can be more easily extended, utilized, or replicated if the underlying representation of the programs it creates is the same as that used by researchers in other organizations.

b) Future Work: Due to the potentially far-reaching benefits that an ISL would have on the research community on HRI systems, we envision its development as an effort that bridges academia and industry. Academic researchers can jumpstart open-source ISL development. If the popularity of an ISL increased, industrial researchers could greatly benefit the wider HRI community by integrating the ISL into the SDKs of their robots. Companies that build development platforms for their robots may consider enabling their ADEs' resulting programs to be compiled to the ISL representation. Recognizing that we, the authors, embody an academic rather than industrial research perspective, a necessary first step toward commercial integration of an ISL will be to more closely align our vision with that of the industrial HRI community. We aim to learn more about industry-specific challenges, such as the involvement of multiple stakeholders within the development of a robot platform, how an ISL can be constructed so as not to restrict third-party developers from extending a robot platform, and how an ISL can facilitate the passing of design constraints between independent stakeholders.

Existing languages and representations may serve as starting points for creating an ISL. *PRISM* [22] and *UPPAAL* [23] serve as a basis for our notional grammar due to their direct support for verification. *Agent planning programs* similarly allow the specification of agent goals within transition systems, while additionally incorporating a planning domain that enables these programs to be synthesized, or *realized* [24]. Additional languages have emerged from within the *Belief-Desire-Intention* (BDI) paradigm that enable the expression of goals, such as *AgentSpeak* [10] and variants of *CAN* (*e.g.,* [11]). BDI languages have also been used to define semantics for goal life cycles [25]. For supporting predefined and reusable skills within a choice representation, *Goal Skill Networks* offer insight into encoding learned policies within hierarchical goal networks [26]. Additional, non-state-based formalisms are common in robotics (*e.g.,* [27]) and should also be investigated as potential starting points. Bringing an ISL to fruition may involve extending one of these existing representations to adhere to all six guidelines, or using them as inspiration to develop entirely new representations.

References

[1] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "CoSTAR: Instructing collaborative robots with behavior trees and vision," in *Proc. ICRA*, 2017, pp. 564–571.

[2] P. Robotics, "Moveit! studio — developer platform & sdk," 2023, https://picknik.ai/studio/.

[3] M. J.-Y. Chung, J. Huang, L. Takayama, T. Lau, and M. Cakmak, "Iterative design of a system for programming socially interactive service robots," in *Social Robotics*, 2016, pp. 919–929.

[4] G. Huang *et al.*, "Vipo: Spatial-visual programming with functions for robot-IoT workflows," in *Proc. CHI*, 2020, p. 1–13.

[5] E. Pot, J. Monceaux, R. Gelin, and B. Maisonnier, "Choregraphe: a graphical tool for humanoid robot programming," in *Proc. RO-MAN*, 2009, pp. 46–51.

[6] N. Leonardi, M. Manca, F. Paternò, and C. Santoro, "Trigger-action programming for personalising humanoid robot behaviour," in *Proc. CHI*, 2019, p. 1–13.

[7] M. Fox and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains," *JAIR*, vol. 20, pp. 61–124, 2003.

[8] D. E. Smith, J. Frank, and W. Cushing, "The ANML language," in *Proc. ICAPS KEPS Workshop*, 2008.

[9] M. Thielscher, "GDL-III: A description language for epistemic general game playing," in *Proc. IJCAI*, 2017, pp. 1276–1282.

[10] A. S. Rao, "Agentspeak(L): BDI agents speak out in a logical computable language," in *Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996, pp. 42–55.

[11] S. Sardina and L. Padgham, "A BDI agent programming language with failure handling, declarative goals, and planning," *JAAMAS*, vol. 23, pp. 18–70, 2011.

[12] D. F. Glas, S. Satake, T. Kanda, and N. Hagita, "An interaction design framework for social robots," in *Proc. RSS*, 2012, pp. 89–96.

[13] S. Alexandrova, Z. Tatlock, and M. Cakmak, "RoboFlow: A flow-based visual programming language for mobile manipulation tasks," in *Proc. ICRA*, 2015, pp. 5537–5544.

[14] D. Porfirio, A. Sauppé, A. Albarghouthi, and B. Mutlu, "Authoring and verifying human-robot interactions," in *Proc. UIST*, 2018, pp. 75–86.

[15] Y. Cao, Z. Xu, F. Li, W. Zhong, K. Huo, and K. Ramani, "V.Ra: An in-situ visual authoring system for robot-IoT task planning with augmented reality," in *Proc. DIS*, 2019, p. 1059–1070.

[16] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Proc. of UbiComp*, 2015, p. 215–225.

[17] E. Senft, M. Hagenow, R. Radwin, M. Zinn, M. Gleicher, and B. Mutlu, "Situated live programming for human-robot collaboration," in *Proc. UIST*, 2021, p. 613–625.

[18] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT press, 2008.

[19] J. E. Young, T. Igarashi, and E. Sharlin, "Puppet master: Designing reactive character behavior by demonstration," in *Proc. SCA*, 2008, p. 183–191.

[20] M. Quigley, "ROS: an open-source robot operating system," in *Proc. ICRA workshop on open source software*, 2009.

[21] J. Young, K. Ishii, T. Igarashi, and E. Sharlin, "Style by demonstration: Teaching interactive movement style to robots," in *Proc. IUI*, 2012, pp. 41–50.

[22] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. CAV*, 2011, pp. 585–591.

[23] G. Behrmann *et al.*, "Uppaal 4.0," in *Proc. QEST*, 2006, p. 125–126.

[24] G. De Giacomo, A. E. Gerevini, F. Patrizi, A. Saetti, and S. Sardina, "Agent planning programs," *Artificial Intelligence*, vol. 231, pp. 64–106, 2016.

[25] J. Harland, D. N. Morley, J. Thangarajah, and N. Yorke-Smith, "An operational semantics for the goal life-cycle in BDI agents," *JAAMAS*, vol. 28, pp. 682–719, 2014.

[26] S. Patra *et al.*, "A hierarchical goal-biased curriculum for training reinforcement learning," *Proc. FLAIRS*, vol. 35, 2022.

[27] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *Proc. ICRA*, 2014, pp. 5420–5427.