Robot Creation from Functional Specifications

Ankur M. Mehta, Joseph DelPreto, Kai Weng Wong, Scott Hamill,
Hadas Kress-Gazit, and Daniela Rus

Abstract The design of new robots is often a time-intensive task requiring multi-disciplinary expertise, making it difficult to create custom robots on demand. To help address these issues, this work presents an integrated end-to-end system for rapidly creating printable robots from a Structured English description of desired behavior. Linear temporal logic (LTL) is used to formally represent the functional requirements from a structured task specification, and a modular component library is used to ground the propositions and generate structural specifications; complete mechanical, electrical, and software designs are then automatically synthesized. The ability and versatility of this system are demonstrated by sample robots designed in this manner.

1 Introduction

Although robots have become prevalent in academic and industrial applications, there is a knowledge barrier which prevents them from fully integrating into daily life. The creation of robots typically requires deep understanding of the available tools as well as the expertise to combine parts in a way that will achieve some desired behavior. Due to this intensive methodology, a discrepancy often arises between the end users and the robot creators, leading to general-purpose robots being created before a task is fully specified.

The long-term vision is to instead enable the creation of custom personal robots on-demand by encapsulating the needed low-level knowledge into computational tools that can automatically address a user's high-level robotic needs. Typical end users will have a task that they want the robot to perform and an understanding of the task requirements, but may not be able to construct or even assemble integrated programmed electromechanical mechanisms to realize a solution. This paper therefore moves towards a system that can compile a high level behavioral description into a completely fabricable robot design including software.

Ankur M. Mehta, Joseph DelPreto, Daniela Rus
Massachusetts Institute of Technology, Cambridge, MA,
e-mail: mehtank, delpreto, rus@csail.mit.edu

Kai Weng Wong, Scott Hamill, Hadas Kress-Gazit
Cornell University, Ithaca, NY,
e-mail: kw358, sbh92, hadaskg@cornell.edu

This work improves and expands the robot compiler system presented in [18]. Whereas the earlier system required a user to provide the complete structural specification for a robot design, from the selection of components to their connectivity and subsequent geometric layout, the system presented herein accepts a more intuitive functional specification as initial input – a description of the relationships between atomic robot action primitives. The selection of components from a library is then aided by automated filtering, and a geometric layout is generated from basic positional constraints. Connectivity and parameter relationships are automatically derived from the generated geometric layout and robot controller. Furthermore, the system presented here enhances design iteration by allowing users to simulate the robot controller prior to fabrication.

2 Problem Formulation and Contributions

The goal of this work is to be able to generate a complete integrated design and controller for a custom robot on demand. To minimize the requirements on an end user, the inputs to the system must be as high level as possible, with the automation of low-level decisions. The process begins with a functional description of the desired behavior, and ends with a programmed printed electromechanical machine that executes the described task. The presented approach decomposes the robot design process into a series of stages that facilitates rapid prototyping and design iteration.

To define the desired behavior, the user first writes a functional specification, or task specification, in Structured English [14], capturing the requirements and goals of the robot. Though not to the level of natural language programming, this allows a casual user to describe rather than command how the robot should operate through the use of primitive elements called propositions. The ability to decompose a desired task solution into this functional specification is the only technical requirement on the user; no other mechanical, electrical, or computer engineering skills are assumed. The input specification maps directly into Linear Temporal Logic (LTL) formulas, which are the input to a controller synthesis algorithm [4]. If there exists a finite state machine capable of achieving the goals given an adversarial environment, a controller will be generated.

The propositions of this specification are then used to create a structural specification – a specification used to build the custom robot. The structural specification is constructed by mapping the propositions to parameterized robotic building blocks drawn from a robot component library. The system filters the library to recommend components appropriate to each proposition, aiding the user in grounding the specification. In addition, the system assesses the mapped propositions for possible behavioral conflicts, correcting the functional specification as needed. Depending on the components chosen by the user, a single functional specification may generate varied robot configurations that accomplish the same goal. The selected components are then automatically configured and connected, though advanced users can edit and create custom configurations. The result is a parameterized robot design capable of accomplishing the task.

Upon setting desired parameters, the structural specification can be compiled to synthesize printable mechanical fabrication files, electrical wiring instructions, and code for the custom robot. The robot controller generated from the functional specification can be analyzed in simulation, then converted to microcontroller code and directly programmed onto the robot. Once the user has built the robot with the given instructions, the desired behavior will be carried out by the created robot.

The full algorithm for creating a robot from a Structured English specification is described in pseudocode listing 1.

Algorithm 1 Robot creation from functional specifications

```
Input Structured English specification file
Convert Structured English into LTL
FSM ← Compile LTL into controller automaton

L ← Load Component library
G ← ∅                                              ▷ Grounding list
A ← Filter(L, dataConsumers)                       ▷ Actuators
for all a ∈ FSM[actuators] do
    if a requires physical output then
        A* ← Filter(A, hasMechanicalPorts)
    else
        A* ← Filter(A, ¬hasMechanicalPorts)
    end if
    c ← User select from A*
    G := G ∪ {Ground(a, c)}
end for
S ← Filter(L, dataGenerators)                      ▷ Sensors
for all s ∈ FSM[sensors] do
    c ← User select from S
    G := G ∪ {Ground(s, c)}
end for

Component ← Core(FSM)
for all g ∈ G do
    if g.component has MechanicalPort then
        Attach(Component, g.component.structure)
    end if
    Connect(Component, g.component.signal)
end for

Save Component as structural specification
Output fabrication files for Component
```

The particular contributions of this work are:

- a process for grounding the propositions of an LTL specification to components from a design library to generate a user-guided robot configuration,
- automatic generation of a complete structural specification, including user-guided physical layout and automated controller synthesis, for the compilation of integrated robot designs,
- a process to detect potential behavioral conflicts during the proposition grounding process, and to automatically correct the LTL specification accordingly,

- filtering algorithms to simplify user interactions with the robot compiler API,
- an implementation of all of the above into an integrated end-to-end system generating robots from a Structured English task description, and
- sample robots generated using the system.

3 Related work

3.1 Functional specification

There has been an increasing interest in the automatic construction of provably-correct robot controllers from high-level or temporal logic task specifications in the robotics community. These controllers, if successfully synthesized, will behave as specified in the mission statements.

The synthesis and execution of these controllers from temporal logic specifications have been shown by [12, 6]. Groups have since tackled a variety of challenges using these controllers, such as the problem of a changing workspace during controller execution [16, 1], conducting motion planning for robots given temporal goals [2], or generating optimized robot trajectories from temporal logic task specifications [24]. These controllers are also used to control multiple robots [11].

The AI and planning community also has planning languages, from STRIPS [7] to PDDL [17] and more extensions, to create functional specification for machines. Using the planning languages, a problem, or functional specification, is defined and solved with the composition of different actions that each consist of preconditions or post-conditions. The generated plans are similar to the controllers generated from high-level task specifications. In this paper, we use high-level task specifications to generate provably-correct robot controllers out of preference.

The functional specification system for this work stems mostly from [13]. Given a robot model and its environment, controllers that satisfy high-level task specifications are composed automatically. The synthesized controllers also respond to different environment behaviors during controller execution.

3.2 Robot creation

There has previously been substantial work regarding processes to fabricate robots. For example, robots have been developed from 2D processes [3, 19, 22] using a range of materials at many different size scales. Some of these methods have also been applied to rapid prototyping [9], although the design phase typically still requires significant time and expertise. Efforts have been made to automate the decomposition of 3D shapes into 2D fold patterns [5, 15, 23], but these often do not address compliant or kinematic structures. Furthermore, these works remain within the realm of designing desired structures rather than abstracting the design input to a task-based level.

There has also been significant work on modular robotic systems and behavior [26, 25, 21, 10]. While these systems provide substantial configurational flexibility,

modular robots often lack the specialized physical components to address specific behavioral tasks.

This work builds most directly upon the robot compiler presented in [18], which describes a system for generating integrated mechanical, electrical, and software designs for custom robots using a modular component library. The system is extended here by abstracting the user input to a higher level: instead of starting with structural specifications, the user can now begin with a desired behavioral task. Once this high-level task has been processed to create functional requirements, the robot compiler's component library and computational tools are used to generate fully functional origami-inspired foldable robots.

4 Design flow

To facilitate the rapid prototyping of custom robots from a description of desired behavior, the proposed approach here creates a user-friendly environment by providing a suite of integrated tools that break the process described in Algorithm 1 into a series of well-defined, computer-aided stages, illustrated in Figure 1. To detail this process, we will consider the following example:

Example 1. The user wants to build a robot that can conduct a pick-and-place grasper task. When the robot receives a request from the user, it will move to a pick-up location and wait for an object to be presented. The robot will then pick up the object, return to the original position, release the object, and inform the user that it has completed the task.

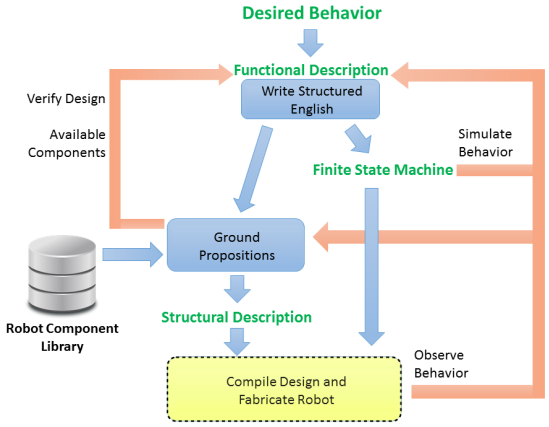


Fig. 1: The system aids robot design by decomposing the procedure into a series of manageable stages. The process can also become iterative at each stage, using feedback from simulation or fabricated devices to encourage rapid prototyping. The dotted yellow box contains the previous work from [18], employed by the system in this paper.

4.1 Behavioral description to functional specification

To create a custom robot from a description of desired behavior, the user starts by writing a mission specification for the task to be conducted. The specification of Example 1 is shown in Figure 2. Using the Linear Temporal Logic MissiOn Planning

```
robot starts with not waitingForObject

# Wait for an object after user summons
waitingForObject is set on userSummons_d and
reset on pickUpObject_m

# Wait at the source for the object
do moveToSource_m  if and only if waitingForObject
# Close gripper when object is detected at source
do pickUpObject_m if and only if seeObject_d and
moveToSource_m
# Indicate completion
do indicateComplete_ud if and only if not
pickUpObject_m and not moveToSource_m
```

Fig. 2: The desired behavior of a robotic grasper can be cast into Structured English, from which a finite state machine is automatically generated and a functional description is naturally extracted.

toolkit (LTLMoP) [8], the user can write a specification in Structured English by first defining different types of binary propositions. These propositions are abstracted from the robot location and actions and its environment. The propositions can be divided into four different types:

- Region propositions: If a map is given to the robot, the map can be decomposed into different regions, with each region being one proposition. During the controller execution, only one of the region propositions is true at any given time, representing the current location of the robot. In Example 1, there are no region propositions for simplicity.
- Sensor propositions: These are propositions abstracted from the robot's surrounding environment. In Example 1, the presence of an object is abstracted into a proposition called "seeObject_d" that is true when an object is observed and false otherwise. Note that even though these propositions describe different sensing capabilities of the robot, they are independent of how this capability is implemented on the custom robot; the propositions specify the functionality of the component rather than the actual structural component. In Example 1, "userSummons_d" is also a sensor proposition.
- Actuator propositions: These are propositions abstracted from the possible actions of the robot. In Example 1, activating an actuator proposition "moveToSource_m" specifies that the robot should move to the pick-up location (and if the proposition is deactivated it implies the robot should be at the drop-off location). As with the sensor propositions, these actions are functional rather than structural and therefore independent of implementation; for example, a robot with legs may move differently than a robot with wheels. In Example 1, "pickUpObject_m" and "indicateComplete_ud" are also actuator propositions.
- Custom propositions: These are propositions that are directly linked to neither robot sensing nor actions but that are necessary for specifying more complex behaviors. In Example 1, once "userSummons_d" becomes true, the proposition "waitingForObject" becomes and remains true until "pickUpObject_m" is true.

Using these propositions, the user can follow the grammar outlined in [14] to write a specification, and a robot controller can be generated with the synthesis al-

gorithm in [4]. This correct-by-construction robot controller, in the form of a finite-state machine, will be automatically generated using the toolkit if the mission statement from the user is feasible regardless of how the environment behaves. The finite state machine generated for Example 1 can be found in the supplementary material at `http://web.mit.edu/mehtank/www/isrr2015/`.

Once a controller is created, the finite state machine can be evaluated with an integrated simulation engine in which sensors can be interactively triggered and the proposition states can be visualized. A sample visualization of the engine is shown in Figure 3 for Example 1. This facilitates an iterative process in which the user can immediately see how the robot would behave and adjust the specification or functional requirements accordingly.

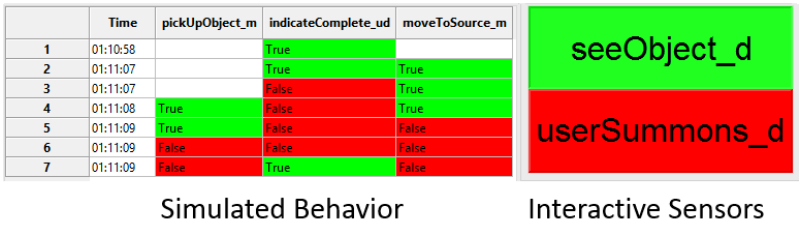


Fig. 3: The generated finite state machine can be simulated to ensure desired behavior and encourage iterative design. Here, the behavior of a pick-and-place grasper is being simulated. The simulation displays the number of state changes, the time changes occur, and the status of the propositions.

4.2 Functional description to structural specifications

4.2.1 Grounding

Once a functional description, or specification, of the robot has been obtained from the behavioral description, a physical instantiation of the robot that achieves the target task must be determined. In particular, the action and sensing tasks to be performed by the robot can now be grounded to available robot components to generate a structural description of the robot.

To ground the functional propositions to structural components, the grounding editor in the toolkit, modified from [8] for robot creation as shown in Figure 4, first retrieves the library of possible modular robotic components from the robot compiler [18]. These components include basic building blocks as well as previously constructed assemblies, each designed to implement a specific behavior. Each component in the library encapsulates the design and fabrication information relevant to the component, including the mechanical structure, the electrical properties and the software of the component, and is parameterized to allow customizability.

The components are currently divided into three types: mechanical components, which require constructed structural elements to interface an electromechanical transducer with the environment; device components, which are discrete devices

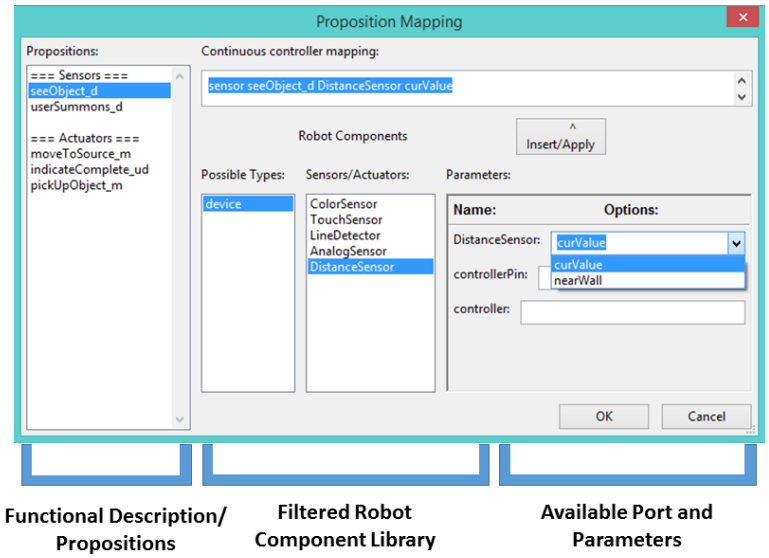


Fig. 4: The functional description can be converted to a structural description by selecting modular components from a robot library for each action and sensor proposition. Filtered lists of possibilities are automatically provided, and the user can choose to customize them by setting parameters or simply accept the default values.

with a completely self-contained action; and UI components, which are purely virtual components that include smartphone interface elements such as sliders or toggle switches.A user can specify desired possible component type(s) for proposition by suffixing its name; based on that suffix, the grounding editor displays a filtered list of allowable components. For the case of Example 1, a list of mechanical actuator components will be shown for the proposition `pickUpObject_m`, while a list of device and UI actuator components will be shown for the actuator proposition `indicateComplete_ud`.

The user can then ground each proposition to one of the available components on the filtered list to obtain the desired actions and sensing capabilities. It is possible that no library component can adequately satisfy a particular proposition, indicating that the proposition is too complex given the existing contents of the library. In this case, the user can either modify the original specification to decompose that proposition into simpler constructs, or create a new component to satisfy the needed behavior. In the latter case, the user would write a new functional specification to define the needed component in terms of simpler propositions, and adding the successful design back into the library.

When a component is chosen from the library, a list of the component parameters is also presented so the user can customize the component if needed. Through this process, a mapping is created between the propositions and the robot components. More specifically, each actuator and sensor proposition is assigned to a port on a component.

It is possible to map more than one proposition to a given component. To ensure conflicting behaviors do not occur, the mapping interface evaluates each mapped proposition and modifies the original functional specification to include mutual exclusions of propositions mapped to the same component. The user is then informed of this modification so they can ensure that the desired behavior is still achieved. The grounding editor creates a close-loop design process by providing feedback to the user through amendments to the functional specification based on the structural specification. With the grounding editor, not only the functional specification affects the structural specification, but the structural specification set by the user also changes the functional specification.

When the compiler processes the design, it will automatically insert multiplexers as appropriate to ensure that the correct command is sent to the component. It should also be noted that some of the components employ analog signals; to meet the boolean requirements of the generated controller, analog sensors get thresholded before becoming inputs to the finite state machine, while the binary actuator commands from the controller are scaled to a user-specified analog value before being applied to the device.

Some possible groundings of propositions in Example 1 are shown in Figure 5. The conversion of functional description to structural specification is aided by the toolkit but is ultimately chosen by the user; the user asserts control over the design according to personal preference and task-specific requirements, such as environmental consideration and component availability. Since there are often many components which can be grounded to the same proposition, many different robots can result from the same functional description. For example, a human-generated input may be mapped to a button, a microphone, or a UI element, while an indicator action may be mapped to a light, a buzzer, or a flag waver. This approach simplifies and guides the robot design process for novice users without restricting expert users.

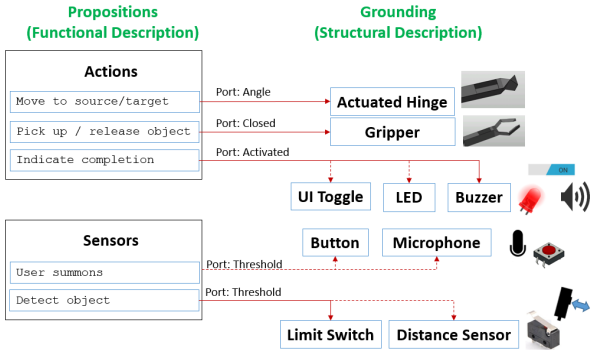


Fig. 5: For each functional proposition needed for the robotic grasper, there are numerous possible robot components in the library that can be used for implementation. Here, a few such options are shown and the solid lines indicate those chosen for the current experiment.

4.2.2 Mechanical connections

A structural specification also requires the geometric layout of the physical components into a single integrated electromechanical device. Though the design space of geometric configurations can get intractably large, the system once again aids a

novice user by presenting a reduced set of options to handle general cases. An expert user can bypass the filter and create arbitrary mechanical connections constrained only by available interface points designed to limit component collision.

The mechanical-type components preferentially presented for grounding are designed to mostly fit into a rectangular prism bounding box. This allows for physical composition by tiling the selected components into orthogonal regions. The user can select whether a particular component belongs in the front, back, left, right, or center of the robot; the system then iterates through the full list of mechanical components and appends them onto the core controller module, growing the robot as it goes. Components with parameterized dimensions get scaled to fit the entire collection.

In a similar manner, the remaining non-mechanical device-type components then get mounted on any exposed face of the robot. The user can specify whether the device should be facing forwards, backwards, left, right, up, or down, and the system will mount the device onto the respective structures assembled in the previous step.

4.3 Integrated robot fabrication

Once the complete structural specifications have been generated, the robot compiler processes the modular design into design files for the complete robot [18], producing mechanical fabrication files, electrical wiring instructions, and microcontroller code.

The mechanical structure is fabricated using an origami-inspired cut-and-fold process: the generated fabrication file gets sent to a desktop vinyl cutter to be cut from a 2D sheet of plastic, and the user then follows the folding instructions and the generated wiring instructions to fabricate the robot. Finally, the automatically generated robot software can be loaded onto the main controller, ranging from low level drivers to the implementation of the finite state machine created from the LTL specification.

The robot can then simply be powered on to achieve the task specifications initially provided by the user.

5 Assumptions, Generalizations and Guarantees

5.1 Functional specification

We consider functional specifications where the lexicon, or the set of words that can be used, corresponds to physical and computational components of a possible robot. In this paper, specifications are written in Structured English which has a deterministic and well defined grammar [14]. This grammar allows for the specification of safety constraints, goals and conditional expressions. The design flow described in this paper easily generalizes to functional specifications given in natural language as long as the natural language utterance can be represented formally using propositions that can be grounded, such as the work in [20]. In future work, we will leverage natural language processing tools to enrich the expressivity and ease of use of the design tools.

5.2 Grounding propositions to computational and physical components

The current process allows a user to map more than one proposition to a single component. This may be done by the user if different propositions are intended to dictate different behaviors for the same component. As an example, the user may ground the propositions `secureObject` and `releaseObject` to the same physical component `Gripper`, with the intent of having the gripper open when `releaseObject` is true and having the gripper close when `secureObject` is true. However, a problem may arise if the structured English specification allows both `secureObject` and `releaseObject` to be true simultaneously, resulting in contradictory commands to the same physical component, which will prevent the robot from achieving the desired behavior and may damage the robot.

The grounding interface addresses this issue by evaluating the grounded propositions and assessing which propositions (if any) have been grounded to the same component. The system then appends mutual exclusion clauses to the structured English specification so that the propositions may not both be true simultaneously, alerting the user to the change. This process is demonstrated with the *Fetch Robot* case study described in Section 6. When parsing the specification into complete robot designs, the compiler will automatically insert multiplexers into the data flow in order to ensure that the component receives the correct command.

5.3 Robot behavior guarantees

The controller generated for the robot is correct-by-construction, which means that provided the assumptions made by the user hold during controller execution and the user chose an appropriate grounding scheme, the robot will behave as expected.

6 Case studies

6.1 Pick-and-place grasper

A sample robot made using this system is a robotic grasper. In this case a user desires a robot which, when prompted by the user, moves to a starting location and waits for an object; when an object is detected, it grasps it, moves to a target location, and notifies the user. This behavior can be written in a Structured English description as shown in Figure 2, and the generated finite state machine can be examined via simulation as shown previously in Figure 3. There are a variety of ways in which this can be grounded to generate a structural description, and a few such possibilities along with the one chosen here are depicted in Figure 5; custom propositions represent internal state that do not become grounded in robot components. During the process of choosing robot components from the library, various parameters such as arm length or gripper size can be set by the user according to their task's environment and restrictions.

Once the grounding is complete, the robot compiler generates a fold pattern along with electrical instructions and Arduino code. The resulting robot is shown in Fig-

ure 6. After uploading the generated code, the arm demonstrates the desired behavior. When the user claps, the robot moves to the source location and waits for an object. It grasps the object upon detection, moves to the target location, releases the object, and indicates completion using a buzzer. Various metrics regarding the robot's performance as well as its design process are summarized in Table 1.

Specification 1 Linear Temporal Logic Specification for Example 1

$$
\begin{aligned}
&\neg\pi_{waitingForObject} \wedge \\
&\square((\bigcirc \pi_{userSummons\_d} \wedge \neg\pi_{pickUpObject\_m}) \rightarrow \bigcirc \pi_{waitingForObject}) \wedge \\
&\square(\pi_{pickUpObject\_m} \rightarrow \neg \bigcirc \pi_{waitingForObject}) \wedge \\
&\square((\pi_{waitingForObject} \wedge \neg\pi_{pickUpObject\_m}) \rightarrow \bigcirc \pi_{waitingForObject}) \wedge \\
&\square((\neg\pi_{waitingForObject} \wedge \neg \bigcirc \pi_{userSummons\_d}) \rightarrow \neg \bigcirc \pi_{waitingForObject}) \wedge \\
&\square(\pi_{waitingForObject} \leftrightarrow \bigcirc \pi_{moveToSource\_m}) \wedge \\
&\square((\pi_{seeObject\_d} \wedge \pi_{moveToSource\_m}) \leftrightarrow \bigcirc \pi_{pickUpObject\_m}) \wedge \\
&\square((\neg\pi_{pickUpObject\_m} \wedge \neg\pi_{moveToSource\_m}) \leftrightarrow \bigcirc \pi_{indicateComplete\_ud})
\end{aligned}
$$

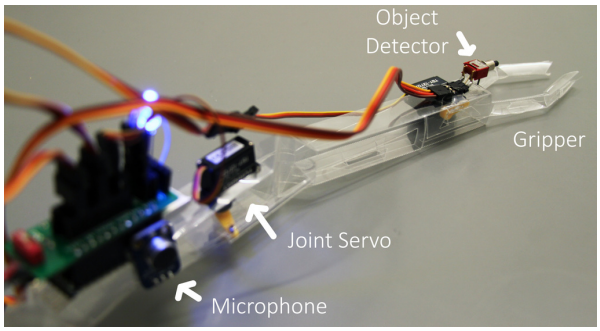


Fig. 6: A pick-and-place robotic grasper was designed using the presented system, starting with a desired behavior and ending with an inexpensive, rapidly manufactured, functional prototype.

Table 1: Performance of robotic grasper

Metric	Result
Approximate design time	30 min
Approximate fabrication time	30 min
Approximate cost	25 USD
Mass	49.4 g
Maximum actuated joint angle	±35 deg
Gripper strength (on 1.5 cm object)	100 mN
Maximum gripper opening	110 mm

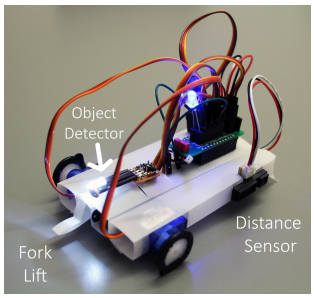
6.2 Fetch robot

A second example robot is a mobile robot with an attached manipulator for retrieving an object placed along a path. The desired behavior is to follow a path until the object is reached, secure the object, continue following the path until the goal is reached, release the object, and indicate completion. This behavior can be written using the Structured English of LTLMoP as shown in Figure 7.

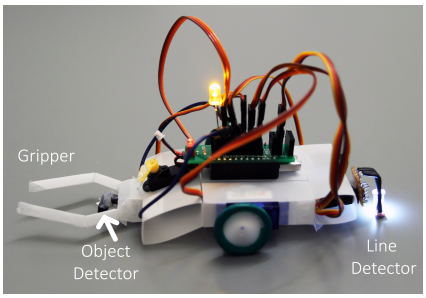
Fig. 7: The desired behavior of a path-following object fetcher can be defined using Structured English. The highlighted statements are necessary to enforce a mutual exclusion condition on propositions grounded to the same physical component, and are automatically generated and added to the behavioral specification.

```
robot starts with false

# Follow the path to get to the object and to move with the object
followPath is set on (not atObject_d and (not secureObject_md or
releaseObject_md)) or (secureObject_md and not atGoal_du) and reset on
(atObject_d and releaseObject_md) or (atGoal_du and releaseObject_md)

# Follow the path
do leftForward_md if and only if (not onPath_d) and followPath
do rightForward_md if and only if onPath_d and followPath

# Grasp object when reached and release it when at goal
do secureObject_md if and only if (atObject_d or secureObject_md) and
not atGoal_du
do releaseObject_md if and only if atGoal_du

# Indicate when object is grasped
do indicateHaveObject_du if and only if secureObject_md

# Indicate when task is complete
do indicateComplete_md if and only if (atGoal_du and releaseObject_md)

if you are activating secureObject_md then do not (releaseObject_md)
if you are activating releaseObject_md then do not (secureObject_md)
```

Fig. 8: Two different robots made with the system which both achieve the desired task of following a path to retrieve an object.



(a) This robot is a line follower and detects the object using a distance sensor.



(b) This robot is a wall follower and detects the object using a touch sensor.

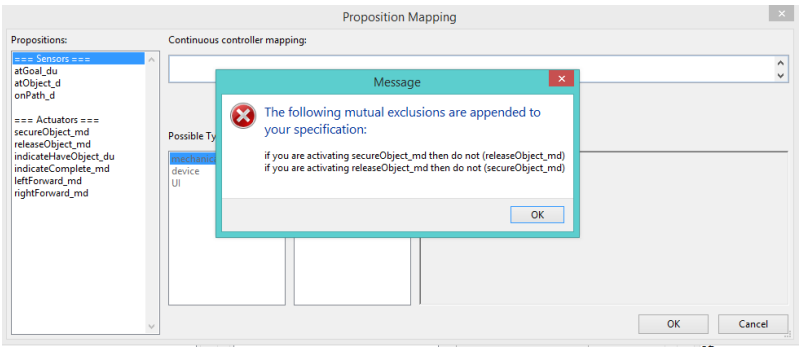


Fig. 9: The mapping process alerts the user when multiple propositions are mapped to the same component. In this case, the propositions `releaseObject_md` and `secureObject_md` have been mapped to the same component port.

To demonstrate the versatility and potential for rapid prototyping, two different sets of groundings were implemented. The chosen components are enumerated in

Table 2, and the completed robots can be seen in Figure 8. Some metrics regarding the performance as well as design of these robots are summarized in Table 3. In both cases there are two propositions, `releaseObject_md` and `secureObject_md`, that are both mapped to the same port of the same component (`Gripper` or `Forklift` depending on the robot). In addition, the line follower instantiation maps both `leftForward_md` and `indicateComplete_md` to the same wheel servo in order to indicate completion with a "victory dance" behavior, spinning in a circle. LTLMoP detects these potential conflicts and notifies the user while automatically generating additional statements necessary to enforce a mutual exclusion on the relevant propositions. A sample notification is shown in Figure 9. The generated statements are then automatically appended to the functional specification as shown in Figure 7.

Table 2: Two separate sets of groundings implemented for a path following fetch robot.

Functional Proposition	Line Follower	Wall Follower
Move forward and left	Wheel 1	Wheel 1
Move forward and right	Wheel 2	Wheel 2
Detect path	Line detector	Distance sensor
Detect object	Touch sensor	Light sensor
Detect goal	UI toggle switch 1	Microphone
Secure object, release object	Gripper	Forklift
Indicate object secured	UI toggle switch 2	LED
Indicate complete	Wheel 1	Buzzer

Table 3: Performance of path following fetch robots

Metric	**Result**		
	Line Follower	**Wall Follower**	
Approximate design time	30	5	min
Approximate fabrication time	60	45	min
Approximate cost	30	45	USD
Mass	64.1	72.1	g
Speed	11.1	11.0	cm/sec
Maximum gripper opening	45	N/A	mm

Once programmed with the generated code, both robot configurations performed the desired task. In addition, the generated code included calibration routines for the sensors when the robot first begins; it prompts the user to provide the minimum and maximum values for each sensor in turn and thereby determines a suitable threshold value for each sensor for converting the analog readings to boolean variables expected by the state machine. For example, the line follower will be placed over white and then over black, and the wall follower will be placed near the wall and then far from the wall. This also grants the user some runtime control over the robot behavior; for example, they can adjust how close the robot stays to the wall by adjusting the positions provided during calibration. Note that the line following robot design also includes UI elements; in this case, the user can use the provided Android app, which will automatically communicates with the generated robot via Bluetooth and display the appropriate user interface (in this case, two toggle switches).

7 Conclusion

In this paper, we present an approach to building and controlling a custom on-demand printable robot from a Structured English functional description, with an end-to-end integrated system implementing the above. This addresses a number of problems often faced by robot designers. Previously, despite the synthesis of a verified robot controller from a task specification, a mission may fail if existing robots are not suitable for the task. On the other hand, constructing custom robots to accomplish a desired behavior requires experience, expertise, tools, and resources.

The work presented here now allows users to start with a vision and follow system-generated recommendations to create a robot to execute that task. The user need not be experienced with robot creation or engineering principles, thus allowing even casual users access to these custom robots; advanced creaters still benefit from design automation. The correct-by-construction controller extends guarantees to the created robot, ensuring a successful mission provided that certain constraints are met. These guarantees coupled with online simulation simplify the design-build-test iteration loop and could easily allow for sophisticated design requirements such as building safety constraints into the robot. With the system demonstrated herein, functional and structural specifications can be matched to each other, allowing for the creation of on-demand robotic solutions for physical tasks.

This paper inspires a number of further research avenues addressing relaxing and avoiding such constraints, while expanding the autonomy provided by the compiler system. The system can be extended to integrate analog signals in a more automated manner for a richer behavioral design space. In addition, more complex functionality can be implemented by enabling a many-to-many mapping between propositions and components; though the compiler supports such a topology, determining mutual exclusion conditions is necessary to ensure provably correct constructions. Finally, a natural language input parser can allow greater flexibility in task specifications, potentially allowing more fine-grained recommendations of components for grounding through an analysis of the circumstances in which the proposition appears.

Supplementary material A video demonstrating the system presented in this paper can be found at `http://web.mit.edu/mehtank/www/isrr2015/`.

Acknowledgements This work was funded in part by NSF ExCAPE and grants #1240383 and #1138967 and NSF Graduate Research Fellowship 1122374, for which the authors express thanks.

References

[1] Ayala AIM, Andersson SB, Belta C (2012) Probabilistic control from time-bounded temporal logic specifications in dynamic environments. In: Robotics and Automation (ICRA), pp 4705–4710

[2] Bhatia A, Kavraki LE, Vardi MY (2010) Sampling-based motion planning with temporal goals. In: Robotics and Automation (ICRA), pp 2689–2696

[3] Birkmeyer P, Peterson K, Fearing RS (2009) Dash: A dynamic 16g hexapedal robot. In: Intelligent Robots and Systems (IROS), IEEE, pp 2683–2689

[4] Bloem R, Jobstmann B, Piterman N, Pnueli A, Sa'ar Y (2012) Synthesis of Reactive(1) designs. J Comput Syst Sci 78(3):911–938
[5] Demaine ED, Tachi T (2009) Origamizer: A practical algorithm for folding any polyhedron
[6] Fainekos GE, Kress-Gazit H, Pappas GJ (2005) Temporal Logic Motion Planning for Mobile Robots. In: Robotics and Automation (ICRA), pp 2020–2025
[7] Fikes RE, Nilsson NJ (1971) Strips: A new approach to the application of theorem proving to problem solving. In: Proc. of the 2nd IJCAI, London, UK, pp 608–620
[8] Finucane C, Jing G, Kress-Gazit H (2010) LTLMoP: Experimenting with language, Temporal Logic and robot control. In: IROS, pp 1988–1993
[9] Hoover AM, Fearing RS (2008) Fast scale prototyping for folded millirobots. In: Robotics and Automation (ICRA), 2008., IEEE, pp 886–892
[10] Hornby G, Lipson H, Pollack J (2003) Generative representations for the automated design of modular physical robots. Robotics and Automation, IEEE Transactions on 19(4):703–719
[11] Karaman S, Frazzoli E (2008) Complex mission optimization for multiple-UAVs using linear temporal logic. In: American Control Conference, Seattle, WA, pp 2003 – 2009
[12] Kloetzer M, Belta C (2008) A Fully Automated Framework for Control of Linear Systems from Temporal Logic Specifications. IEEE Trans Automat Contr 53(1):287–297
[13] Kress-Gazit H, Fainekos GE, Pappas GJ (2007) Where's Waldo? Sensor-Based Temporal Logic Motion Planning. In: Robotics and Automation (ICRA), pp 3116–3121
[14] Kress-Gazit H, Fainekos GE, Pappas GJ (2008) Translating Structured English to Robot Controllers. Advanced Robotics 22(12):1343–1359
[15] Lang R (2012) Origami design secrets : mathematical methods for an ancient art. A K Peters/CRC Press
[16] Livingston SC, Prabhakar P, Jose AB, Murray RM (2013) Patching task-level robot controllers based on a local mu-calculus formula. In: Robotics and Automation (ICRA), pp 4588–4595
[17] McDermott D, et al (1998) PDDL – the planning domain definition language – version 1.2. Tech. rep., Yale Center for Computational Vision and Control
[18] Mehta AM, DelPreto J, Shaya B, Rus D (2014) Cogeneration of mechanical, electrical, and software designs for printable robots from structural specifications. In: Intelligent Robots and Systems (IROS)
[19] Onal C, Wood R, Rus D (2013) An origami-inspired approach to worm robots. Mechatronics, IEEE/ASME Transactions on 18(2):430–438
[20] Raman V, et al (2013) Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language. In: Robotics: Science and Systems IX, Technische Universität Berlin, Berlin, Germany, June 24 - June 28, 2013
[21] Romanishin J, Gilpin K, Rus D (2013) M-blocks: Momentum-driven, magnetic modular robots. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on, pp 4288–4295
[22] Shimoyama I, Miura H, Suzuki K, Ezura Y (1993) Insect-like microrobots with external skeletons. Control Systems, IEEE 13(1):37–41
[23] Tama Software Ltd (2015) Pepakura designer. `http://www.tamasoft.co.jp/pepakura-en/`, [Online; accessed 01-Apr-2015]
[24] Wolff EM, Topcu U, Murray RM (2014) Optimization-based trajectory generation with linear temporal logic specifications. In: Robotics and Automation (ICRA), pp 5319–5325
[25] Yim M, Duff D, Roufas K (2000) PolyBot: a modular reconfigurable robot. In: Robotics and Automation (ICRA), vol 1, pp 514–520 vol.1
[26] Yim M, Shen WM, Salemi B, Rus D, Moll M, Lipson H, Klavins E, Chirikjian G (2007) Modular self-reconfigurable robot systems [grand challenges of robotics]. Robotics Automation Magazine, IEEE 14(1):43–52