Towards Language-Based Verification of Robot Behaviors

Anthony Cowley and Camillo J. Taylor

***Abstract*— The management of finite resources is central to many robot behaviors. Some robotic systems must maintain invariants regarding the disposition of feet for balancing, others have grippers for manipulating their environments, while yet others must respect strict rules governing the usage of objects in the environment. Yet the specifics of such resource management responsibilities are almost universally locked behind opaque controllers whose lack of type information greatly impedes rigorous static analysis. We present an application of dependent type theory and linear logic for the static analysis of robot behavior programs that manage both robot and environment state, with a worked assembly task example. This approach offers static, formal guarantees with respect to safety requirements attached to primitive actions, as well as introspection of expected state at each step of a scripted sequence of actions allowing for the automatic generation of dynamic, sensor-based, runtime verification of successful execution.**

I. Introduction

A mobile manipulator such as that shown in Figure 1 requires an array of software to successfully function. Sensor data must be filtered and integrated, controllers must be tuned, and plans must be computed. As the primary perception and control components of the system come on-line, one is able to construct an abstraction layer exposing the capabilities of the system in semantically meaningful units that build upon motion primitives guided by sensor feedback. For instance, the platform shown is capable of actuation to open, close, and position its gripper in 3D space (X and Y axes define the work surface, or table, with the positive Z axis extending upwards).

A combination of perception and workspace design – including a cache from which new blocks may be retrieved – enable the robot to work with notched building blocks designed to fit together to facilitate the assembly of small structures. A high-level programming interface for the robot in a C-like language looks like this,

```
void get_from_cache(void);
void move_arm(int x, int y, int z);
void open_gripper(void);
```

These primitive actions may be sequenced to effect an assembly operation,

```
get_from_cache();
move_arm(1,2,3);
open_gripper();
get_from_cache();
move_arm(4,2,0);
get_from_cache();
```

This work was not supported by any organization.

Anthony Cowley and Camillo J. Taylor are with the GRASP Laboratory at the University of Pennsylvania, Philadelphia, PA 19104, USA {acowley,cjtaylor}@@seas.upenn.edu

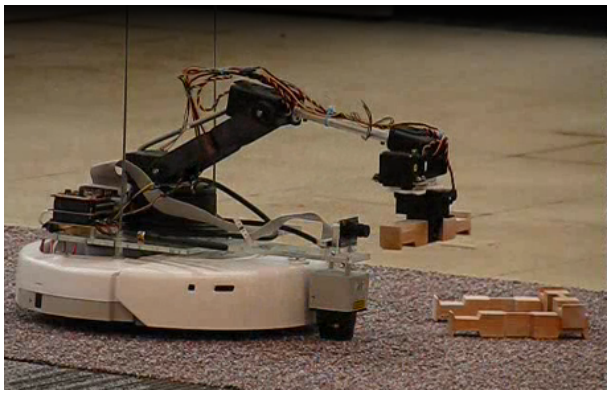

Fig. 1. Mobile manipulator platform stacking blocks.

This program is unfortunate: it type checks but is riddled with bugs.

II. Richer Types

The above assembly script starts out with a bug: what is the behavior if the gripper is holding something when the robot is told to get a new block from the cache? The first block is placed at coordinates $(1, 2, 3)$, 3 units above the table. Is there anything below that location to support the new block? Is there already something at that location that the new block will collide with? Moving on, the next bug is another non-empty-gripper bug, because the programmer failed to call `open_gripper()` after moving the arm to the second location before going back to the cache for a new block.

These potential errors are due to underspecification of the primitive actions. The `get_from_cache` action is defined as being valid for *all* states the world might be in, and is silent regarding how it might change the state of the world. Intuitively, the reality of this function is that, even though it does not compute anything of interest, this primitive action has associated pre- and post-conditions:

$$gripperIsEmpty \Rightarrow \texttt{get\_from\_cache} \Rightarrow gripperHolds$$

Turning this inside out, we can say that,

$$\texttt{get\_from\_cache} : gripperIsEmpty \rightarrow gripperHolds$$

meaning that `get_from_cache` converts a *gripperIsEmpty* (abbreviated *empty* from now on) to a *gripperHolds* (abbreviated *holds*). However, *empty* is not a number, or some infinitely copyable abstraction, it is a finite resource of the system, and ceases to exist when acted upon by the `get_from_cache` function. We may view *empty* as a kind

of *certificate* that may be used just once. In the case of `get_from_cache`, it is exchanged for another single-use certificate, *holds*. These certificates may be used to ensure that we do not attempt to pick up a block while already holding one.

Functions that change the world rather than just the robot state, by placing a block on the table for example, should similarly reflect that change in their types. If the original state of the world is Γ (a set of blocks), and the robot is to place a block at coordinates (x, y, z), then we wish for a `place_block` function whose type is something like,

$$block_{xyz} \notin \Gamma \rightarrow holds \rightarrow (\Gamma \cup block_{xyz}) \otimes empty$$

where $\otimes$ is a kind of product, or conjunction.

This function type requires a witness that the target block is not already in the environment Γ, consumes a certificate that the robot is currently holding a block, and produces a new environment with the block added on to Γ along with a certificate that the gripper is *empty*. The linearity of the certificate premises of this type may be reflected by positing an axiom for this primitive action that makes use of a linear logic provability relation, $\vdash\!\!\!-$,

$$\begin{aligned} &let\ B\ :=\ block_{xyz}\ in \\ &B \notin \Gamma \rightarrow (\Gamma \otimes holds \vdash\!\!\!- (\Gamma \cup \{B\}) \otimes empty) \end{aligned}$$

This form of reasoning allows us to ascribe a rich propositional language to simple imperative programs. The appeal lies in the fact that writing down a full logical specification for a complicated structure assembly, including all the intermediate states of its construction, is unwieldy. Even for the simple cases considered here, dozens of propositions are needed to describe what we believe to be true about simple assembly scripts. We thus aim to view the language in which our actions are written as an encoding of a rigorous formal proof. This is accomplished by attaching highly expressive types to the basic actions our system can perform in order to produce an axiomatization of the system. Critically, the types of the end-product programs are inferred so that the programmer is not required to leave the familiarity of imperative scripting.

III. Linear Logic

To formalize these intuitive notions of finite resource management, we present a fragment of linear logic as introduced by Girard [1]. To emphasize a computational interpretation of the logic, we work with intuitionistic linear logic in a natural deduction style, similar to the presentations found in [2], [3], [4], to give readers a taste of the features most useful for the verification of robot behaviors. As a point of departure, the central feature of linear logic as it applies here is that while the following proof is valid in classical and constructive logics, it is **not** valid in linear logic.

$$\dfrac{\dfrac{\dfrac{[A]^1 \qquad \dfrac{[A]^2 \qquad [A \rightarrow B]^3}{\vdash B}}{\vdash A \times B}}{A, A, A \rightarrow B \vdash A \times B}\ {\scriptstyle [1,2,3]}}{A, A \rightarrow B \vdash A \times B}\ \text{contraction}$$

Following tradition, premises are written above a line separating them from derived conclusions, square brackets are used to indicate assumptions, and superscripts are used to track the discharge of those assumptions. The entailment, or provability, relation, denoted $\vdash$, here corresponds to either the classical or constructive provability relations. The examples here serve to contrast linear logic with both classical and constructive logics, so the reader unfamiliar with constructive logics may consider $\vdash$ as the provability relation of classical logic.

The *contraction* rule available in classical and constructive logics may be interpreted as stipulating that a single premise, A, may be used to conclude both A, by *identity*, and B, by the implication $A \rightarrow B$.

In contrast, the linear logic provability relation, denoted $\vdash\!\!\!-$, allows neither contraction nor weakening (essentially, a statement that additional hypotheses may always be added to a derivation). If we distinguish linear implication from classical implication by using the notation $A \multimap B$ to denote an implication that consumes its argument, then the linear provability relation enforces that the hypotheses A and $A \multimap B$ may *either* be used to conclude A or B, but not both. An example of this style of reasoning is that, with an empty-gripper certificate in hand, we may pick up a block, which will result in a certificate verifying that we now hold a block, but the original empty-gripper certificate will have been consumed.

This notion of a virtual, one-time-use safety certificate has many familiar analogs in the average person's day-to-day life. Consider the predicament of the vending machine user: he must choose to either retain his dollar, or exchange it for a soda. We may represent the dollar as A, the soda as B, and the vending machine as a function with type $A \rightarrow B$. Thinking of the function as an implication, both classical and constructive logics permit use of the hypothesis A any number of times, enabling us to keep the dollar in our pocket *and* feed it to the vending machine to obtain a soda. Linear logic, on the other hand, forces us to reason somewhat differently.

The vending machine itself is logically defined by,

$$\dfrac{A \vdash\!\!\!- B}{\emptyset \vdash\!\!\!- A \multimap B}\ \multimap\text{-I}$$

which states that since an A may be consumed to produce a B, we have a linear implication $A \multimap B$ (with $\emptyset$ indicating that there are no assumptions for the conclusion of this proof). Note that derivations like this are not the same as setting a boolean flag. If we have two derivations, we may freely combine them to build a new derivation. The distinction is that resources are tracked through composition: if two derivations each consume an A, then the composition

of those derivations will require the availability of two A propositions.

The vending machine logic is actually a special case of the *linear implication* introduction rule, denoted $\multimap$-I when used in proofs. This rule describes how linear implications may be formed in a general context, or list of hypotheses, represented as either Γ or Δ in the following.

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B} \multimap\text{-I}$$

The linear implication implemented by the vending machine results in a customer-vending machine system (with Γ standing in for the customer) described by,

$$\frac{\Gamma \vdash A \qquad \dfrac{\Gamma \vdash A \qquad \emptyset \vdash A \multimap B}{\Gamma \vdash B} \multimap\text{-E}}{\Gamma \vdash A \;\&\; B} \&\text{-I}$$

The linear implication elimination rule, $\multimap$-E, is used to produce a B which is combined with A to form the proposition $A \;\&\; B$ using the *with* introduction rule, denoted &-I. This introduction rule requires that one be able to conclude both A and B from the context Γ, but not necessarily at the same time. This is reflected in the availability of the two elimination rules,

$$\frac{\Delta \vdash A \;\&\; B}{\Delta \vdash A} \;\&\text{-E}_1 \qquad \frac{\Delta \vdash A \;\&\; B}{\Delta \vdash B} \;\&\text{-E}_2$$

These two rules mean that if we have a *with*, also known as an additive conjunction, then we may choose which component to use.

Contrast this with the multiplicative conjunction, denoted $\otimes$, corresponding to the standard *and* logical connective. The elimination rule for this connective, $\otimes$-E, requires that we be able to derive our conclusion, C, from a context extended by *both* components of the product. Formally,

$$\frac{\Gamma \vdash A \otimes B \qquad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C} \otimes\text{-E}$$

In the application presented here, the linear logic provability relation is used to work with single-use certificates denoting facts of a fleeting nature (e.g. "block A is *on* block B"), and to keep track of blocks placed in the environment in order to statically detect collisions. These properties of the system are produced and consumed over the course of execution, enabling one to reason about *sequences* of actions by interpreting those actions as steps in a proof built from the above inference rules.

IV. Formalization in Coq

The Coq proof assistant [5], based on a core calculus known as the Calculus of Inductive Constructions (CIC) [6], is a tool for developing formal mathematical proofs, specifications, and certified programs. Since everything is formalized in Coq using CIC, all logic and proof development is actually implemented as type checking. The power of this system is that once one trusts the theory, and a very small kernel implementing this theory, any CIC expression that type checks represents a legitimate proof.

Underpinning this style of proof is an embrace of the Curry-Howard isomorphism, which may be generalized to state that a type is a proposition, and a program having that type is a proof of the proposition. As an example, one may pose the proposition that there exists a set, `Bool`, of boolean values. A proof of this proposition is a witness of this claim, such as `true`. Thus one may read the typed program term `true : Bool` (pronounced, "`true` has type `Bool`") as a pair of proof and proposition.

More intriguingly, one may slightly abuse the fact that the standard notations for logical implication and function type stand in coincidence to say that a function of type `Int→Int` is a proof of the proposition that `Int` implies `Int`. Such an implication requires a program that, given an integer, produces an integer.

A. Dependent Types

Reasoning about typed programs, one may view a higher order function expecting an argument of type `Int→Int` as parameterized over a proof of this claim. Presumably this parameterization serves some purpose: usually an intent to *use* the proof object (note the computational interpretation asserting itself). However, there is a logical side to such a parameterization, too. It may be that a function makes no active use of a particular proof object supplied as an argument, but simply wishes that there be a witness of the specified type. For the case of `Int→Int`, this is unlikely, but consider a type such as `(n:ℕ)→even n→ ℕ` that may be used to annotate, say, a function that divides a natural number – type $\mathbb{N}$ – by two. The idea here is that we would like to write a "divide by two" function, `divByTwo`, on natural numbers such that,

```
∀ (n:ℕ), 2 * divByTwo n = n.        (1)
```

The aim of this type is to preclude the function having to return well-typed values for arguments that are not divisible by two. Without the second parameter, whose type identifies a subset of the natural numbers, the function author would need to decide what natural number to return for an odd argument, such as 3. The richer type means that the function implementation may satisfy the desired specification, Eq. 1, without requiring some *ad hoc* protocol specification of how naturals not divisible by two are handled.

The example `divByTwo` function has what is known as a *dependent type* [7]: a type that may depend on values. Note in the above example that the first argument to the function is of type $\mathbb{N}$, but we have bound a name, `n`, to the value passed to the function. Subsequent parts of the type may now refer to the value passed to the function, and these references may be used to assert properties of this value. In this case, the program implementing the `divByTwo` function – i.e. the proof of the type – need not worry that `n` is not divisible by two, because a well-typed application of the function must include a *proof* of `even n`, where `even` is an inductively defined family indexed by a natural number.

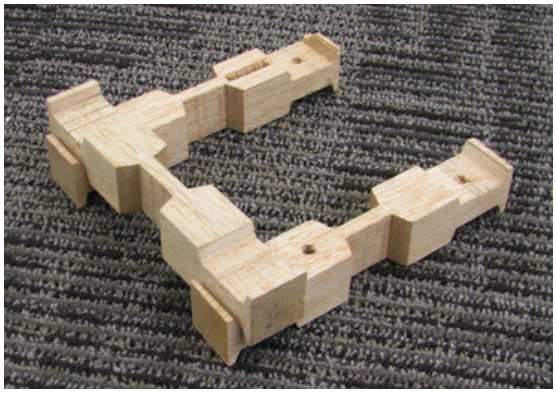

Fig. 2. A three-block structure.

B. Embedding Linear Logic

The embedding of linear logic in Coq is based on that described by Power and Webster [8]. We define the linear logic consequence relation as an inductive type whose constructors correspond to the introduction and elimination rules shown in Section III.

Given a type of linear propositions, `LinProp`, we can define the intuitionistic linear logic consequence relation as a `Prop` (corresponding to a logical proposition in Coq's type system) indexed by a list of `LinProp`s corresponding to the context and a `LinProp` corresponding to the conclusion. In Coq syntax, this is,

```
Inductive LinCons :
   list LinProp → LinProp → Prop :=
```

The constructors of this type take some number of parameters to assemble an application of the `LinProp` type constructor, here with the infix ⊢ notation used earlier. For example, the multiplicative conjunction introduction is implemented as,

```
TimesRight : ∀ A B Γ Δ,
  (Γ ⊢ A) → (Δ ⊢ B) → (Γ ++ Δ ⊢ A ⊗ B)
```

The constructors for the `LinCons` type family define a way of producing values of Coq's `Prop` type. In Section II, we said that the assembly actions we intend to deal with do not produce interesting computational values. Because the real-world actions of the robot are distinct from values computed within the programming language, we focus on the things that are available to the programming language: the types.

We therefore use the `LinCons` indexed type family to produce types, and demand that our assembly programs produce values of these types. Similar to the `divByTwo` example, we will be writing functions that take parameters whose value is solely *logical*, not computational. If we have a value of type Γ ⊢ `A` and a value of type Δ ⊢ `B`, then we may use the `TimesRight` constructor to produce a value of type Γ ++ Δ ⊢ `A` ⊗ `B`.

V. Assembly Tasks

Given that we are working with virtual values whose only purpose is to carry informative types, we bootstrap the system by introducing axioms that define the actions our robot is capable of performing. These axioms are the kernel of the logical specification of the application. Our goal is to program the mobile manipulator platform shown in Figure 1 to produce structures composed of interlocking blocks as shown in Figure 2.

A. Data Types and Linear Facts

We define a coordinate system for these blocks such that block placements are aligned with the X and Y axes that define the ground plane. With this convention established, we refer to the more negative and more positive ends of blocks as their "left" and "right" ends. To capture a block's position, (x, y, z), we define a 3D point data type with integer coordinates,

```
Inductive Point : Set :=
  point : ℤ → ℤ → ℤ → Point.
```

The `Block` data type builds on `Point` to also capture the block's orientation.

```
Inductive Block : Set :=
  block : Point → Orientation → Block.
```

In order to ensure that we do not attempt to force two blocks to occupy the same volume of space, we maintain a set data structure, type `BlockEnv`, recording all the `Block`s known to be in the environment. The environment is a fact recorded in the linear context as, `env : BlockEnv → LinProp`.

Other propositions involving a single block are `clearL : Block → LinProp`, indicating that a block's left end is clear, `clearR : Block → LinProp` indicating that a block's right end is clear, and `table : Block → LinProp` indicating that a block is on the table. The `clear`$\star$ propositions will let us work with blocks that may not be physically capable of supporting stacked blocks.

Blocks may be vertically stacked in two ways: they may be stacked directly on top of each other, `on : Block → Block → LinProp`, or one block may bridge two perpendicular blocks as shown in Figure 2. The bridging configurations are indicated by the linear proposition, `bridgesL : Block → Block → Block → LinProp` indicating that the first `Block` bridges the left ends of the second and third `Block`s, and `bridgesR` for the support blocks' right ends.

A final structural property is required to verify bridging configurations: the bridge supports must be parallel, aligned, and a particular distance apart. We encode linear safety certificates regarding the suitability of two potential foundation blocks as `safeBridgeL : Block → Block → Block → LinProp` to indicate that the first `Block` may safely bridge the left ends of the second two, and `safeBridgeR` for the right ends of the support blocks.

Finally, we track the gripper's binary state using `holds : LinProp` and `empty : LinProp` indicating whether the gripper is currently holding a block, or if it is empty.

We will not be directly producing values using these `LinProp` constructors. Instead, we will solely rely on a small number of axioms that define the primitive aspects of

our robotic platform. By virtue of being inhabitants of the `LinProp` type, all these logical facts may take part in the `LinCons` relation that we will use to reason about the state of our system.

B. Axiomatized Assembly

To begin, the manipulator is able to fetch new blocks from a cache of blocks,

```
Axiom newBlock : [empty] ⊢ holds.
```

This axiom states that, given a linear context consisting of `empty`, indicating that the gripper is not holding anything, one may conclude that the gripper now `holds` a block. This corresponds to the real-world action of the mobile manipulator driving to the parts cache and picking up a fresh block.

Adding a block to the environment is our only way of building anything, but there are variations that we must consider. In each case, placing a block requires that the new block not collide with any block already known to be in the environment. Note that this fact is not linear: it is not a resource, but a proposition in the native constructive logic of Coq. We check a block against an environment with,

```
Definition collisionFree E X :=
  ∀ Y, Y ∈ E → noOverlap X Y.
```

where `noOverlap` is a function producing a proposition that there is a separating plane between two blocks based on integer inequalities involving block dimensions. The types are inferred in the definition of `collisionFree`, but it has type `BlockEnv → Block → Prop`. It is a claim that `Block` *X* does not overlap any block *Y* in environment *E*.

We may finally write down the axiom that defines our ability to place a block in the workspace,

```
Axiom put : ∀ X Y Z E,
collisionFree E X →
([holds ⊗ env E] ⊢
  empty ⊗ clearL X ⊗ clearR X ⊗
  env (add E X) ⊗
  (table X &
   clearL Y ⊗ clearR Y ⊸ on X Y &
   safeBridgeL X Y Z ⊸ bridgesL X Y Z &
   safeBridgeR X Y Z ⊸ bridgesR X Y Z)).
```

The `put` axiom states that if block *X* is `collisionFree` in environment *E*, then a linear context consisting of the facts that the gripper `holds` a block and that the current environment is, in fact, *E*, implies that the system state may transition such that the gripper is `empty`, block *X* is `clear` on both ends, the current environment is now *E* augmented by block *X*, and a fact describing *X*'s configuration.

This last item is an additive conjunction of the various facts that may describe a newly placed block: the programmer may conclude any *one* of the options. The newly placed block may be,

- a foundational block placed on the table
- directly on top of another block, *Y*, given the linear assumptions that both ends of *Y* are `clear`
- bridging the left or right ends of blocks *Y* and *Z* given that such a bridging configuration is safe.

As this axiom handles all the ways a block may be placed, we prove several specialized placement lemmas to handle the various cases in the style of [8]. These lemmas, an example of which is reproduced here, represent the first public interface of the system specification to be exposed to users.

```
(* Put block X on block Y. *)
Lemma puton : ∀ X Y E,
  collisionFree E X →
  ([env E ⊗ holds ⊗ clearL Y ⊗ clearR Y] ⊢
   empty ⊗ clearL X ⊗ clearR X ⊗
   env (add E X) ⊗ on X Y ).
```

These specialized lemmas are useful units of composition, but they still require some work to use. Power and Webster [8] use a little bit of Coq's automation facilities, embodied in the *Ltac* language for defining proof-writing tactics, to make theorem proving somewhat easier, but we take this further by demanding that semantically meaningful actions should correspond to a single interaction in Coq's interactive proving mode. These proof writing tactics absolve the user from having to perform repetitive context manipulations or derive facts like collision safety that result from decidable integer inequality proofs and set operations.

In general, each lemma is accompanied by a tactic that rewrites the current proof goal so that the lemma may be applied and automatically discharges assumptions of the lemma. Note that this scripting of the proof assistant is not skipping over any of the obligations for rigor that we desire, it is merely automating the process of specifying the incremental logical steps needed to produce a desired result.

C. Program or Proof

The choice of using Coq's interactive theorem proving mode – wherein the user enters commands that update a sequent tracked and displayed by Coq – rather than its more traditional program writing facilities reflects the way the system will be used. While one may write functions that ostensibly *compute* with the detailed types we are building through the application of the various constructors, we opt for writing down lemmas and theorems which we then must prove to Coq's satisfaction. The two tasks are intimately connected, but the interaction with a proof assistant gets at precisely the value Coq brings to the robotics programming domain.

A formal, machine-checked proof contains nothing more than a finite list of steps needed to derive the conclusion from the assumptions. In contrast, an informal proof, intended as communication between humans, often elides steps of logical reasoning in favor of documenting key aspects of the proof state on the path from assumptions to conclusion. This difference presents the central contribution of this work: a robot program is a specification of the operations the robot is to perform, while the proof assistant meticulously tracks the state of the process. To be clear, the dynamic state of a

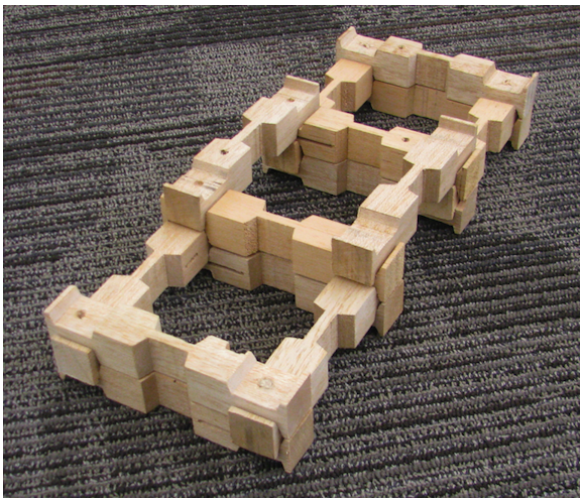

Fig. 3. A fourteen-block structure comprising two towers with spanning bridges.

proof is the type of the composition of the proof steps up to any given point.

This tracking of state, missing from typical imperative scripts, allows the proof checker to guarantee that the program/proof is free of defects that would guarantee its failure. Finally, we note that the proof state is a highly granular safety predicate as it reflects the dynamic state of the system, rather than committing to a safe subset of the system's configuration space at the outset. The safe states of the system depend on what has come before: safety is causal.

D. Putting It All Together

We are now ready to consider the statically verifiable program needed to generate the assembly script corresponding to the structure shown in Figure 3. This structure is built from two towers, each consisting of six blocks, with two spanning blocks bridging the gap between the towers. To name the key components of the structure, we refer to the blocks directly supporting the bridging spans as `PeakLeft` and `PeakRight` for the peaks of the left and right towers, while the spanning blocks are named `SpanA` and `SpanB`.

We rely extensively on helper tactics as described at the end of Section V-B: `get_from_cache` wraps the `newBlock` axiom; `put_on_table` wraps the `puttb` lemma (itself a specialization of the general `put` axiom); and `bridge{L,R}` wrap specializations of `put` for placing blocks that bridge pairs of support blocks.

The target structure shows a large amount of repetition in the tower assemblies that we naturally wish to abstract using a function. Rather than writing a long series of statements of the form,

```
get_from_cache();
bridgeL(X,Y,Z);
```

we will work with a function that executes the task of adding a layer to a tower any number of times. We will call this function, implemented as a tactic, `build_tower`, and parameterize it by the integer (X,Y) coordinates of the tower center and the natural number of levels to add to the tower. We will use the names introduced above for blocks serving particular roles so that we may refer to them by name in the program rather than having to type in their coordinates repeatedly.

```
build_tower(2,3,2);
build_tower(6,3,2);
get_from_cache();
bridgeL(SpanA, PeakLeft, PeakRright);
get_from_cache();
bridgeR(SpanB, PeakLeft, PeakRight);
```

This program, which is enough to drive the system shown in Figure 1, builds one tower centered at (X,Y) coordinates (2,3) with two layers above its foundation, and another at (6,3). The two spanning blocks are retrieved from the cache and used to bridge the gap between the towers.

Enough Coq automation has been provided that this C-like syntax can be simply translated into the following verifiable Coq program,

```
Lemma TowerBridge : [empty, env ∅] ⊢ ⊤.
Proof.
  intros.
  build_tower 2 3 2%nat.
  build_tower 6 3 2%nat.
  use_aliases [PeakLeft, PeakRight].
  get_from_cache.
  bridgeL SpanA PeakLeft PeakRright.
  get_from_cache.
  bridgeR SpanB PeakLeft PeakRight.
Qed.
```

The linear context at the end of this program is far from `void`: it contains 34 facts about the world, including an environment proposition recording the 14 placed blocks. This context may be inspected to verify that it trivially satisfies propositions about the resultant structure. For instance, the propositions,

```
bridgesL SpanA PeakLeft PeakRight
```

and

```
bridgesR SpanB PeakLeft PeakRight
```

are elements of the ending context.

The imperative program dressed up with the `Lemma` syntax is used to prove a trivial proposition, during the course of which Coq verifies that each step is legal. Before ending the interactive proof session, the accumulated linear type context is extracted as it represents the most specific proposition proved by the original program. This specific type is used to annotate the elaborated proof to provide a modular unit for reuse in future constructions.

VI. Related Work

Systems and languages for the assembly of shapes and structures have seen increasing attention recently [9], [10], [11], [12], [13], [14]. These works focus on operational aspects of getting pieces into place to form a structure, rather than physical properties of those structures or the systems doing the assembling. Napp [15] describes the composition of guarded programs for organization, but the composition of guard clauses is not targeted by a static feasibility analysis.

There is also a connection to the extensive research done on using temporal logics for motion specifications, some highlights of which include [16], [17], [18], and [19]. These works are primarily concerned with ensuring that a robot, or team of robots, is always in a valid state, and/or eventually transitions into a particular set of

accepting states. To this end, specifications are written in a temporal logic that refers to elements of an identified state space.

Our approach differs from these works in several ways. First, we do not aim to automatically identify a witness of a proposition, but instead seek to determine what proposition a given program is witness to. Given a putative proof, we attempt to identify the most specific theorem. Second, the specification of system state is tremendously burdensome when dealing with all the logical facts one can derive from a particular action. The domains we are interested in are both *dynamic* and *infinite*. Rather than manually identify a desired system state, we emphasize the identification of complex configurations through the composition of a core set of actions with known properties. Third, we focus on giving the platform designer tools to enable application specific formalization in the form of language metatheory. For instance, to prove collision freedom we made use of a `noOverlap` function that produced a disjunction of possible separating planes between two blocks. In order to automate proofs about collision safety, it is helpful to know that `noOverlap` is a commutative relation, a fact readily proved within Coq.

These differences set this work apart from model checking approaches more commonly found in robotics today. Model checking is, in a sense, the flip side of the approach taken by this paper. Constructing the specification – a sentence in some logic – is laborious and error prone, it should be automated. The approach shown here synthesizes the specification from the program while simultaneously verifying that the given program is a proof of the derived specification.

VII. Conclusions and Extensions

We have shown examples of language-based verification for programs controlling a single robot with a single gripper. Many of the facts proved by this system may seem trivial: verification that the gripper is empty before it is used to pick up another block could probably be performed by a human looking over the program before running it. But, setting aside the benefits of automating the verification of even simple properties, the system described extends immediately to systems with, say, four grippers, merely be starting the specification off with a larger initial context. Instead of starting out in an empty environment with an empty gripper, `[empty ⊗ env ∅]`, we can start out with the context, `[empty ⊗ empty ⊗ empty ⊗ empty ⊗ env ∅]`, and use the same assembly lemmas and tactics to verify that our multi-manipulator is never over-extended.

The expressiveness of linear logic is a good fit for robotic systems that frequently must deal with finite resources while exploring infinite domains characterized by the introduction and elimination of dynamic objects over the course of execution. By embedding the necessary logical building blocks in a proof assistant like Coq, the system designer empowers application programmers with a rich toolkit for formal verification of critical system properties.

Acknowledgments

The authors wish to thank Professor Ani Hsieh and the Drexel University SAS Lab for access to their automated assembly platform, and Jean Gallier for the "Girardian turnstile" LaTeX.

References

[1] J.-Y. Girard, "Linear logic," *Theor. Comput. Sci.*, vol. 50, pp. 1–102, January 1987. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(87)90045-4

[2] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*. North, 1990. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.5002

[3] ——, "A taste of linear logic," in *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, ser. MFCS '93. London, UK: Springer-Verlag, 1993, pp. 185–210. [Online]. Available: http://portal.acm.org/citation.cfm?id=645722.666394

[4] T. Braüner, "Introduction to linear logic," University of Aarhus, Tech. Rep. LS-96-6, Dec. 1996. [Online]. Available: http://www.brics.dk/LS/96/6/BRICS-LS-96-6/BRICS-LS-96-6.html

[5] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004. [Online]. Available: http://www.labri.fr/perso/casteran/CoqArt/index.html

[6] T. Coquand and G. Huet, "The calculus of constructions," *Inf. Comput.*, vol. 76, pp. 95–120, February 1988. [Online]. Available: http://portal.acm.org/citation.cfm?id=47724.47725

[7] H. Barendregt, "An Introduction to Generalized Type Systems," *Journal of Functional Programming*, vol. 1, no. 2, pp. 125–154, April 1991.

[8] J. Power and C. Webster, "Working with linear logic in coq," in *The 12th International Conference on Theorem Proving in Higher Order Logics*, 1999.

[9] R. Nagpal, "Programmable self-assembly using biologically-inspired multiagent control," in *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, July 2002.

[10] E. Klavins, "Automatic synthesis of controllers for distributed assembly and formation forming„" in *Proceedings of the IEEE Conference on Robotics and Automation*, May 2002.

[11] E. Klavins, R. Ghrist, and D. Lipsky, "A grammatical approach to self-organizing robotic systems," *IEEE Transactions on Automatic Control*, vol. 51, no. 6, 2006.

[12] J. Werfel and R. Nagpal, "Three-dimensional construction with mobile robots and modular blocks," *Int. J. Rob. Res.*, vol. 27, pp. 463–479, March 2008. [Online]. Available: http://dx.doi.org/10.1177/0278364907084984

[13] J. Werfel, Y. Bar-Yam, D. Rus, and R. Nagpal, "Distributed Construction by Mobile Robots with Enhanced Building Blocks," in *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, May 2006.

[14] S. Yun, M. Schwager, and D. Rus, "Coordinating construction of truss structures using distributed equal-mass partitioning," in *Proc. of the 14th International Symposium on Robotics Research*, Luzern, Switzerland, Aug 2009.

[15] N. Napp and E. Klavins, "Robust by composition: Programs for multi-robot systems," in *International Conference on Robotics and Automation (ICRA10)*, 2010, pp. 2459–66.

[16] M. Antoniotti and B. Mishra, "Discrete event models+temporal logic=supervisory controller: automatic synthesis of locomotion controllers," in *IEEE International Conference on Robotics and Automation*, vol. 2, May 1995, pp. 1441 –1446 vol.2.

[17] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for mobile robots," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, April 2005, pp. 2020–2025.

[18] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[19] M. Kloetzer and C. Belta, "Automatic deployment of distributed teams of robots from temporal logic motion specifications," *IEEE Transactions on Robotics*, vol. 26, pp. 48–61, February 2010. [Online]. Available: http://dx.doi.org/10.1109/TRO.2009.2035776