IEEE Transactions on Robotics, 11-0659, 10.1109 / TRO.2013.2239553

The Motion Grammar: Analysis of a Linguistic Method for Robot Control

Neil Dantam, *Student Member, IEEE,* Mike Stilman, *Member, IEEE*

***Abstract*—We present the Motion Grammar: an approach to represent and verify robot control policies based on Context-Free Grammars. The production rules of the grammar represent a top-down task decomposition of robot behavior. The terminal symbols of this language represent sensor readings that are parsed in real-time. Efficient algorithms for context-free parsing guarantee that online parsing is computationally tractable. We analyze verification properties and language constraints of this linguistic modeling approach, show a linguistic basis that unifies several existing methods, and demonstrate effectiveness through experiments on a 14-DOF manipulator interacting with 32 objects (chess pieces) and an unpredictable human adversary. We provide many of the algorithms discussed as Open Source, permissively licensed software.** [1]

***Index Terms*—Hybrid Control, Control Architectures and Programming, Formal Methods, Manipulation Planning**

I. Introduction

Safety is important for physical robots where failures impose physical costs. Model-based verification helps improve safety. Hybrid systems models present robots with both continuous and discrete dynamics. Continuous dynamics use differential equations. Using software to handle discrete dynamics, however, presents challenges for safety due to the general-case inability to guarantee software performance. We can address this difficulty using *Formal Language* models [28] to syntactically define the system [41]. While prior linguistic methods have focused on finite-state Regular languages, we can describe a broader class of system behavior using the Context-Free language class. Synthesizing results from Discrete Event Systems and Compiler Design [1], we analyze the discrete syntax of hybrid controllers and introduce a new model for discrete dynamics, the Motion Grammar, which provides advantages in representative power and hierarchical design while still maintaining verifiability and efficient online operation.

Linguistic control methods describe the set of discrete paths a system may take. Each path, or language string, is a sequence of abstract symbols representing relevant events, predicates, states, or actions. Explicitly defining this system language lets us algorithmically verify system performance [3, 27]. When this system language is *parsed online*, it defines a control policy enabling response to unpredictable events. The typically used Regular language class is limited to finite discrete state. The Context-Free set provides more descriptive power while maintaining the efficiency and verifiability of Regular languages. In addition, Context-Free Grammars provide a natural representation for hierarchies in the system. Thus, we extend the linguistic control approach to Context-Free Grammars.

This paper analyzes the discrete components of a hybrid robotic system through Formal Language. Our model, the Motion Grammar (Sect. IV), uses Context-Free Grammars to represent and verify discrete dynamics (Sect. V). We demonstrate this approach in the domain of physical human-robot chess (Sect. VI). The linguistic formalization also shows a unifying basis for several alternative representations of discrete dynamics (Sect. VII). The Motion Grammar integrates robotic perception and control, providing theoretical and practical benefits.

There are several advantages to the Context-Free language model used in the Motion Grammar. As with Regular Languages, and unlike other typical language classes (sect. IV-E), we retain verifiability (sect. V-F) and fast reactive response (sect. VI-A). In addition, the grammar representation of a language makes it convenient to specify hierarchies (sect. VI-B), which simplified the construction of our grammar for chess. Fundamentally, a Context-Free language can represent scenarios which a Regular Language cannot (sect. VI-C). This combination of benefits make the Context-Free set a useful model for robot control policies.

II. Related Work

Hybrid Control is a quickly advancing research area describing systems with both discrete, event-driven, dynamics and continuous, time-driven, dynamics. Ramadge and Wonham [41] first applied Language and Automata Theory [28] to Discrete Event Systems. Hybrid Automata generally combine a Finite Automaton (FA) with differential equations associated with each FA control state. This is a widely studied and utilized model [2, 5, 26, 30, 37]. Maneuver Automata use a Finite Automaton to define a set of maneuvers that transition between trim trajectories [20]. In this paper, we model hybrid systems using the Motion Grammar which represents continuous dynamics with differential equations and discrete dynamics using a Context-Free Grammar (CFG) [8], providing benefits in computational power and hierarchical specification while still allowing offline verification and efficient online control [10]. Thus we provide a hybrid systems model which builds on existing approaches in useful ways.

The Motion Description Language (MDL) is another approach that describes a hybrid system switching though a sequence of continuously-valued input functions [4, 29]. This string of controllers is a *plan* whereas the Motion Grammar is a *policy* representing the robot's response to any event.

This work was supported by NSF grants CNS1146352 and CNS1059362.
The authors are with the Robotics and Intelligent Machines Center in the Department of Interactive Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA. email: ntd@gatech.edu, mstilman@cc.gatech.edu

[1]Many algorithms discussed in this paper implemented in our Motion Grammar Kit: http://www.golems.org/node/1224

Model Checking is a technique for verifying discrete and hybrid systems by systematically testing whether the model satisfies a specified property [3, 27]. Typically, model-checking uses a finite state model of the system. However, there are algorithms to check Context-Free systems as well [17, 19]. We describe the specific language classes for which this is possible in sect. V-F. Approaches such as [18, 34, 36] use Linear Temporal Logic (LTL) to formally describe uncertain multi-agent robotics by a finite state partitioning of the 2D environment. We adopt a discrete representation more suitable to high dimensional spaces; our manipulation task uses a 14-DOF robot and 32 movable objects making complete discretization computationally infeasible.

There is a large body of literature on grammars from the Linguistic and Computer Science communities, with a number of applications related to robotics. Fu did some early work in syntactic pattern recognition [21]. Han, et al. use attribute graph grammars to parse images of indoor scenes by describing the relationships of planes in the scene according to production rules [25]. Koutsourakis, et al. use grammars for single view reconstruction by modeling the basic shapes in architectural styles and their relations using syntactic rules [35]. Toshev, et al. use grammars to recognize buildings in 3D point clouds [44] by syntactically modeling the points as planes and volumes. B. Stilman's Linguistic Geometry applies a syntactic approach to deliberative planning and search in adversarial games [43]. Rawal, et al. use a class of Sub-Regular Languages to describe robotic systems [42]. These works show that grammars are useful beyond their traditional role in the Linguistic, Theoretical, and Programming Language communities. Our approach applies grammars to online control of robotic systems.

In the context of safe human-robot interaction, [13] demonstrates safe response of a knife-wielding robot based on collision detection when a human enters the workspace. Other approaches to safe physical interaction between humans and robots are surveyed by [14], and [23] suggests specific methods for different types of safety. The Motion Grammar builds on such methods by providing both task-level guarantees and a common structure to combine these existing techniques.

Other studies have developed implementations for our experimental domain of robot chess. [32] describes a specially designed robot arm and board. [45] developed a robot chess player using a specialized analytical inverse kinematics. [38] describes a new robot arm and perception algorithms to play chess on an unmodified board. Instead of focusing on chess play, we use the context of this physical human-robot game to demonstrate the Motion Grammar. We present a general approach implemented on a existing robot arm using general kinematics methods. Furthermore, we provide features and safeties beyond game-play and manipulation.

III. Background

The Motion Grammar (MG) is a formalism for designing and analyzing robot controllers. It is a computational analogue to formal grammars for computer programming languages. Theoretical results for programming languages are directly applicable to MG making it possible to prove correctness. This paper introduces an implementation of MG and analyzes these guarantees. First, we briefly review formal grammars. For a thorough coverage of language and automata theory, see [28].

A. Review of Grammars and Automata

Grammars define *languages*. For instance, C and LISP are computer programming languages, and English is a human language for communication. A formal grammar defines a formal language, a set of strings or sequences of discrete *tokens*.

Definition 1 (Context-Free Grammar, CFG): $G = (Z, V, P, S)$ where Z is a finite alphabet of symbols called *tokens*, V is a finite set of symbols called *nonterminals*, P is a finite set of mappings $V \mapsto (Z \cup V)^*$ called *productions*, and $S \in V$ is the *start symbol*.

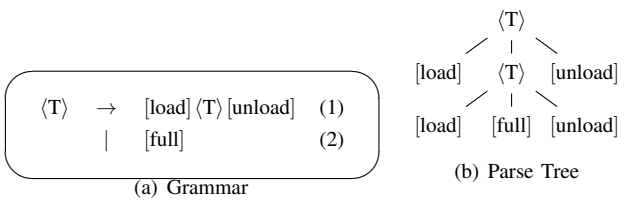
The productions of a CFG are conventionally written in Backus-Naur form. This follows the form $A \to X_1 X_2 \ldots X_n$, where A is some nonterminal and $X_1 \ldots X_n$ is a sequence of tokens and nonterminals. This indicates that A may *expand* to all strings represented by the right-hand side of the productions. The symbol ε is used to denote an empty string. For additional clarity, nonterminals may be represented between angle brackets $\langle\rangle$ and tokens between square brackets $[]$.

Grammars have equivalent representations as *automata* which *recognize* the language of the grammar. In the case of a Regular Grammar – where all productions are of the form $\langle \mathrm{A} \rangle \to [\mathrm{a}] \langle \mathrm{B} \rangle$, $\langle \mathrm{A} \rangle \to [\mathrm{a}]$, or $\langle \mathrm{A} \rangle \to \varepsilon$ – the equivalent automaton is a Finite Automaton (FA), similar to a Transition System with finite state. A CFG is equivalent to a Pushdown Automaton, which is an FA augmented with a stack; the addition of a stack provides the automaton with memory and can be intuitively understood as allowing it to count.

Definition 2 (Finite Automata, FA): $M = (Q, Z, \delta, q_0, F)$, where Q is a finite set of *states*, Z is a finite alphabet of *tokens*, $\delta : Q \times Z \mapsto Q$ is the *transition function*, $q_0 \in Q$ is the *start state*, $F \in Q$ is the set of *accept states*.

Definition 3 (Acceptance and Recognition): An automaton M *accepts* some string σ if M is in an accept state after reading the final element of σ. The set of all strings that M accepts is the *language* of M, $\mathfrak{L}(M)$, and M is said to *recognize* $\mathfrak{L}(M)$.

Regular Expressions [28] and Linear Temporal Logic (LTL) [3] are two alternative notations for finite state languages. The basic Regular Expression operators are concatenation $\alpha\beta$, union $\alpha|\beta$, and Kleene-closure α^*. Some additional common Regular Expression notation includes $\overline{\alpha}$ which is the complement of α, the dot (.) which matches any token, and $\alpha?$ which is equivalent to $\alpha|\varepsilon$. Regular Expressions are equivalent to Finite Automata and Regular Grammars. LTL extends propositional logic with the binary operator *until* $\cup$ and unary prefix operators *eventually* $\Diamond$ and *always* $\Box$. LTL formula are equivalent to Büchi automata, which represent *infinite* length strings, termed ω-Regular languages. We can also write ω-Regular Expressions by extending classical Regular expressions with infinite repetition for some α given as α^ω. These additional notations are convenient representations for finite state languages.



(a) Grammar (b) Parse Tree

Fig. 1. Example Context-Free Grammar for a load/unload task and parse tree for string "[load] [load] [full] [unload] [unload]"

Any string in a formal language can be represented as a *parse tree*. The root of the tree is the start symbol of the grammar. As the start symbol is recursively broken down into tokens and nonterminals according to the grammar syntax, the tree is built up according to the productions that are expanded. The production $A \rightarrow X_1 \ldots X_n$ will produce a piece of the parse tree with parent A and children $X_1 \ldots X_n$. The children of each node in the parse tree indicate which nonterminals or tokens that node *expands* to in a given string. Internal tree nodes are nonterminals, and tree leaves are tokens. The parse tree conveys the full syntactic structure of the string.

An example CFG and parse tree are given in Fig. 1 for a loading and unloading task. In production (1), the system will repeatedly perform [load] operations until receiving a [full] token from production (2). Then the system will perform [unload] operations of the same number as the prior [load] operations. This simple use of *memory* is possible with Context-Free systems. Regular systems are not powerful enough.

While grammars and automata describe the structure or *syntax* of strings in the language, something more is needed to describe the meaning or *semantics* of those strings. One approach for defining semantics is to extend a CFG with additional *semantic rules* that describe operations or actions to take at certain points within each production. Additional values computed by a semantic rule may be stored as *attributes*, which are parameters associated with each nonterminal or token, and then reused in other semantic rules. The resulting combination of a CFG with additional semantic rules is called a *Syntax-Directed Definition* (SDD) [1, p.52].

B. Hybrid Dynamical Systems

Hybrid Dynamical Systems combine discrete and continuous dynamics; this is a useful model for digitally controlled mechanisms such as robots. The discrete dynamics of a hybrid system evolve as discrete state changes in response to *events*. The continuous dynamics evolve as continuous state varies over *time*. We define a hybrid system as,

Definition 4: A hybrid system is a tuple $F = (\mathscr{X}, \mathscr{Z}, \mathscr{U}, Q, Z, \delta, \rho)$ where,

$\mathscr{X} \subseteq \Re^m$	continuous state
$\mathscr{Z} \subseteq \Re^n$	continuous observation
$\mathscr{U} \subseteq \Re^p$	continuous input
Q	set of discrete state
Z	set of discrete events
$\delta : Q \times \mathscr{X} \times \mathscr{U} \mapsto \mathscr{X} \times \mathscr{Z}$	continuous dynamics
$\rho : Q \times Z \mapsto Q$	discrete dynamics

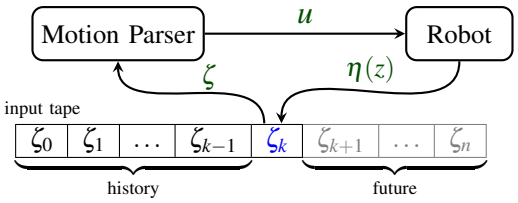


Fig. 2. Operation of the Motion Grammar.

IV. The Motion Grammar

A. Motion Grammar Definition

The Motion Grammar (MG) is a Syntax-Directed Definition expressing the language of interaction between agents and real-world uncertain environments. In this paper, the agent is a robot and the example language represents physical human-robot chess (Sect. VI).

MG tokens are system states or discretized sensor readings. MG strings are histories of these states and readings over the system execution. Like SDDs for programming languages, the MG must have two components: *syntax* and *semantics*. The syntax represents the ordering in which system events and states may occur. The semantics defines the response to those events. The MG uses its syntax to decide from the set of system behavior and semantics to interpret the state and select continuous control decisions. This paper focuses on the *s*yntax of MG, its expressivity, and formal analysis of MG languages.

The Motion Grammar represents the operation of a robotic system as a Context-Free language. The grammar is used to generate the *Motion Parser* which drives the robot as shown in Fig. 2.

Definition 5 (Motion Grammar): The tuple $\mathscr{G}_M = (Z, V, P, S, \mathscr{X}, \mathscr{Z}, \mathscr{U}, \eta, K)$ where,

Z	set of events, or tokens
V	set of nonterminals
$P \subset V \times (Z \cup V \cup K)^*$	set of productions
$S \in V$	start symbol
$\mathscr{X} \subseteq \Re^m$	continuous state space
$\mathscr{Z} \subseteq \Re^n$	continuous observation space
$\mathscr{U} \subseteq \Re^p$	continuous input space
$\eta : \mathscr{Z} \times P \times \mathbb{N} \mapsto Z$	tokenizing function
$K \subset \mathscr{X} \times \mathscr{U} \times \mathscr{Z} \mapsto \mathscr{X} \times \mathscr{U} \times \mathscr{Z}$	set of semantic rules

Definition 6 (Motion Parser): The Motion Parser is a program that recognizes the language specified by the Motion Grammar and executes the corresponding semantic rules for each production. It is the control program for the robot.

From Def. 5, the Motion Grammar is a CFG augmented with additional variables to handle the continuous dynamics. Variables Z, V, P, and S are the CFG component. Spaces $\mathscr{X}$, $\mathscr{Z}$, and $\mathscr{U}$ are for the continuous state, measurement, and input. The tokenizing function η produces the next input symbol for the parser according to the sensor reading and the position within the currently active production. The semantic rules K describe the continuous dynamics of the system and are contained with the productions P of the CFG. Using these discrete and continuous elements, the combined Motion Grammar $\mathscr{G}_M$ explicitly defines the *Hybrid System Path*.

Definition 7 (Hybrid System Path): The *path* of a system defined by Motion Grammar $\mathscr{G}_M$ is the tuple $\Psi = (x, \sigma)$ where,

$x : t \mapsto \mathscr{X}$ continuous trajectory through $\mathscr{X}$
$\sigma \in \mathfrak{L}\{\mathscr{G}_M\}$ discrete string over Z

Though the focus of this paper is on the discrete portion of this hybrid system, we include the continuous components in the definition for three reasons. First, we want to define discrete events based on continuous variables (sect. V-B). Second, we can define functions for the continuous input $\mathscr{U}$ at appropriate positions as semantic rules within grammar (sect. V-D). Third, we provide conditions on the grammar and continuous system path (sect. V-E) that permit discrete reasoning about correctness (sect. V-F).

B. Application of the Motion Grammar

There are two phases where we apply the Motion Grammar to a robotic system: first as a model for *offline reasoning* and second for *online parsing*. The properties of Context-Free languages provide guarantees for each of these phases. Offline, we can always verify correctness of the language (sect. V-F) and there are numerous algorithms [1, 16, 39, 39] for automatically transforming the grammar into a parser for online control. Online, the parser controls the robot. The structure of CFLs guarantees that online parsing is $O(n^3)$ in the length of the string [16], and with some restrictions on the grammar [1, p.222], parsing is $O(n)$ – constant at each time step, a useful property for real-time control.

Online parsing is illustrated in Fig. 2. The output of the robot z is discretized into a stream of tokens ζ for the parser to read. The history of tokens is represented in the parser's internal state, i.e. the stack and control state of a PDA. Based on this internal state and the next token seen, the parser decides upon a control action u to send to the robot. The token type ζ is used to pick the correct production to expand at that particular step, and the semantic rule for that production uses the continuous value z to generate the input u. Thus, the Motion Grammar represents the language of robot sensor readings and translates this into the language of controllers or actuator inputs.

C. Time and Semantics

Next we describe the linguistic properties of the Motion Grammar that arise from the online parsing of the system language. While a translating parser such as a compiler is typically given its input as a file, a Motion Parser must act token-by-token continually driving the system. This temporal constraint restricts the ability of the Motion Parser to *lookahead* and *backtrack*. Thus, we cannot apply an arbitrary Syntax-Directed Definition to an online system but are instead restricted on the type of parser we may use and the allowable ordering of attribute semantics. We now consider the issues of discrete vs. continuous time, selection of productions during parsing, and computation of attributes.

1) Discrete vs. Continuous Time: The continuous dynamics of a system may be modeled and controlled in either continuous or discrete time. For the purpose of modeling, these representations are functionally equivalent. Discrete time models can approximate continuous time by using a sufficiently short timestep, and continuous time models can represent discrete time using timeout events. For implementation on a microprocesser, we must ultimately adopt a discrete time representation; however, this can be obtained by simply discretizing the continuous-time model. The Syntax-Directed Definition of the Motion Grammar can thus be written in either continuous or discrete time as is convenient.

$$\begin{array}{lcl} \langle \mathrm{A} \rangle & \rightarrow & [\mathrm{a}]\,\{u=1\}\,\langle \mathrm{B} \rangle \\ & | & [\mathrm{a}]\,\{u=1\}\,\langle \mathrm{C} \rangle \end{array}$$

(a) Semantically LL(1)

$$\begin{array}{lcl} \langle \mathrm{A} \rangle & \rightarrow & [\mathrm{a}]\,\{u=1\}\,\langle \mathrm{B} \rangle \\ & | & [\mathrm{a}]\,\{u=2\}\,\langle \mathrm{C} \rangle \end{array}$$

(b) Not Semantically LL(1)

Fig. 3. Examples grammar fragments that are and are not Semantically LL(1)

2) Selecting Productions and Semantic Rules: We next compare the Motion Grammar to the LL(1) class of grammars. LL(1) grammars can be parsed by recursively descending through productions, picking the next production to expand using only a single token of lookahead and without backtracking [1, p.222]. While we could satisfy the Motion Grammar's temporal constraint by restricting to an LL(1) grammar, we can relax this restriction slightly. The actual requirement is not that the Motion Parser must immediately know which production it is expanding. Instead, the parser must immediately provide some input to the robot. Thus the parser may use additional lookahead, but only if all productions it is deciding between have *identical semantic rules*. This way, the parser can immediately execute the semantic rule, and use some additional lookahead to figure which production it is really expanding. We describe this property as *Semantically LL(1)*.

Definition 8: A Syntax-Directed Definition is Semantically LL(1) if for all strings in its language, the correct semantic rule to execute can be determined using a single token of lookahead and without backtracking.

Claim 9: A Motion Grammar must be Semantically LL(1).

Proof: The Motion Parser derived from the Motion Grammar, $\mathscr{G}_M$, must be able to immediately provide the system with an input $u \in \mathscr{U}$ in response to each token, and it cannot change the value of inputs already sent. Suppose that $\mathscr{G}_M$ were not Semantically LL(1). This would mean it could use multiple tokens of lookahead or backtrack before deciding on a semantic rule to calculate u. Since u must be known before more tokens are accepted and previous u values cannot be changed, this a contradiction. Thus $\mathscr{G}_M$ must be Semantically LL(1). ■

The Semantically LL(1) property is useful because it allows grammars to be parsed in real-time. Examples of grammars that do and do not satisfy this property are given in Fig. 3. In addition, Fig. 7 is an example of a grammar that is not LL(1) but is Semantically LL(1). This property also permits ambiguous grammars – where multiple parse trees may exist for a given string. This is acceptable because the output of the parser, u sent to the robot, will be the same regardless of which parse tree is selected, and thus the particular resolution of the ambiguity is irrelevant.

When designing our Motion Grammar, we must ensure LL(1) semantics. This is possible with any strictly LL(1) grammar. Non-LL(1) grammars will contain conflicts where two alternative productions may begin with the same token [1, p.222]. If for any conflict, all productions contain the same semantic rules, then the grammar is Semantically LL(1). Generation of efficient parsers for LL(k) and LL(*) grammars is discussed in [39]. If the intended Motion Grammar is not Semantically LL(1), we must either rework the grammar or instruct the parser as to the appropriate precedence levels so that it can resolve any conflicting productions.

3) Attribute Inheritance and Synthesis: Now we consider the structure of the attribute semantics in the Motion Grammar. Attributes are the additional values attached to tokens and nonterminals in an SDD. For the Motion Grammar, these represent the continuous domain values x, z, and u. In our SDD, the attributes of some given nonterminal are calculated from the attributes of other tokens and nonterminals; this introduces a dependency graph into the syntax tree. We must ensure that the dependency graph has no cycles or we will not be able to evaluate the SDD [1, p.310]. The temporal nature of the Motion Grammar constrains the attribute dependencies even further; during parsing, we only have access to information from the past because the future has not happened yet. Attributes can be described as either *synthesized* or *inherited* based on their dependencies. Synthesized attributes depend on the children of the nonterminal while inherited attributes depend on the nonterminal's parent, siblings, and other attributes of the nonterminal itself. The temporal constraint of the Motion Grammar corresponds to a particular class of SDDs called *L-attributed definitions* for the left-to-right dependency chain. A nonterminal X in an L-attributed definition may only have attributes that are synthesized or are inherited with dependencies on inherited attributes of X's parent, attributes of X's siblings that precede it in the production, or on X itself in ways that do not result in a cycle [1, p.313].

Claim 10: A Motion Grammar must have L-attributed semantics.

Proof: We must determine the attributes in a single pass because parsing is online, so the past cannot be changed, and the future is unknown. Let the inherited attributes of nonterminal V be $V.h$, and let its synthesized attributes be $V.s$. For all productions $p = A \rightarrow X_1 X_2 \ldots X_n$, consider the attributes of X_i. While expanding X_i, $A.h$ are known. All $X_j, j < i$ in this production have already been expanded because they represent past action, so $X_j.h$ and $X_j.s$ are also known. However, $X_k, k > i$ represent future actions, so $X_k.h$ and $X_k.s$ are unknown. This also means that $A.s$ is unknown because its value may depend on $X_k.h$ and $X_k.s$. Consequently, $X_i.h$ may only depend on $A.h$, $X_j.h$, and $X_j.s$. $X_i.s$ may depend on attributes from its children because they will be known after X_i has been expanded. These constraints on attributes synthesis and inheritance correspond to L-attributed definitions. ■

D. Languages, Systems, and Specifications

The Motion Grammar models and controls a robotic *system*. Often during controller design, there is a rigid distinction between what is the plant and what is the controller, and analogously, Fig. 2 shows the Robot and the Motion Parser as separate blocks. However, these are arbitrary distinctions. Consider the case of feedback linearization where we introduce some additional computed dynamics so that we can apply a linear controller. While these additional dynamics may physically exist as software on a CPU, for the purpose of designing the linear controller, they are part of the plant. With the Motion Grammar, we have the same freedom to designate components between the plant and controller in whatever way is most convenient to the design of the overall system.

For linguistic control approaches, there is one critical distinction to make between the language of the *system* and the language for the *model*. The system is the physical entity with which we are concerned: the controller and the robot. The model is the description of how the controller and robot respond; it is a set of mathematical symbols on paper or in a computer program. Both the system and the model can be described by formal languages.

Definition 11: The *System Language*, $\mathfrak{L}_g$, is the set of strings generated by the robot and parsed by the controller during operation.

Definition 12: The *Modeling Language*, $\mathfrak{L}_s$, is the set of strings that describe the operation of controllers and robots.

These languages are related. Each string in the modeling language describes a particular system: a robot and controller. This specification is parsed offline to generate the control program. The system language is parsed online *by the control program*. The Motion Grammar is a modeling language that describes a Context-Free *system*.

We emphasize that the Motion Grammar is not simply a Domain Specific Language or Robot Programming Language [6, p.339] but rather the direct application of linguistic theory to robot control in order to formally verify performance. The language described by the Motion Grammar is that of the robotic system itself.

E. The "Goldilocks"[2] Set

For the problem of robot control, where guarantees on performance and verifiability are necessary, the Context-Free Set used in the Motion Grammar is a convenient rank in the Chomsky Hierarchy of formal language classes. First, Context-Free is strictly more powerful model than the Regular languages. Second and more radically, we propose that it is appropriate to *sacrifice Turing-complete computation* in exchange for certain guarantees. We are willing to make this exchange because failures in physical robotic systems can impose severe physical costs; thus, guaranteed safety and reliability are critically important. These benefits and tradeoffs of the Motion Grammar make it an appropriate model for online robot control.

1) Regular Languages: Context-Free languages offer advantages over Regular languages for robot control. The Regular Languages are the simplest of the commonly-used formal languages classes. Regular languages permit strong guarantees on performance and are often used to model reactive control

[2]English idiom for moderation, i.e. die goldene mitte

systems. A major benefit of these models is the ability to verify system behavior. Context-Free languages extend Regular languages with *memory* in the form of a pushdown stack. In sect. VI-C, we use this memory to implement a limited planner within the purely reactive controller. Even with this additional power, Context-Free models still permit formal verification as we show in sect. V-F. Thus, Context-Free languages are more powerful than regular languages, and still permit guarantees on performance.

2) Turing-Recognizable Languages: The demand that a programmer give up Turing-complete computation for a Context-Free Motion Grammar is a drastic one, but it comes with important guarantees. Turing-recognizable or *Recursively Enumerable* languages are the most powerful class in the Chomsky hierarchy. A Turing-complete computational model is nearly universal among computer programming languages. Even this paper was typeset in the Turing-complete LaTeX language. However, the Turing-complete model, with all its power and generality, has a severe cost: the Halting Problem and Rice's Theorem mean that any nontrivial property of a Turing Machine is unprovable [28, p.188]. For a general, Turing-recognizable language *we can guarantee nothing*.

3) Context-Sensitive: Context-Sensitive languages, which fall between the Context-Free and the Turing-Recognizable sets, are not generally suitable for Real-Time control. The general Context-Sensitive decision problem is PSPACE-Complete, a challenge when online response is needed. Thus, we consider the Context-Sensitive Language class to be an unsuitable model for real-time robotic systems.

4) Context-Free: The Context-Free Language class is an especially useful model for online control of robotics systems. Among the Regular, Context-Free, Context-Sensitive, and Recursively-Enumerable sets, the Context-Free languages provide a balance between power and provability for this problem domain. Online robot control requires an immediate response, and Context-Free languages are always parsable in polynomial time [16]. Physical robots require safety and reliability guarantees to prevent damage or injury, and a Context-Free model can always be verified as we prove in sect. V-F. For these reasons, the Context-Free set provides appropriate benefits with acceptable costs compared these other language classes for representing the discrete dynamics of robotic systems.

V. Grammars for Robots

The Motion Grammar is a useful model for controlling physical robots. In this section, we discuss how to apply grammars to robots and illustrate the points with our sample application of human-robot chess. First, we describe the setup for the chess application. Then we explain tokenization and parsing for robot grammars using this example. Finally, we show the guarantees that are possible with the Motion Grammar.

A. Experimental Setup

To demonstrate the concepts and utility of the Motion Grammar, we developed a sample application of physical, human-robot chess. This application ran on a Schunk LWA3 7-DOF robot arm with a Schunk SDH 7-DOF, 3-fingered hand as shown in Fig. 4. A wrist mounted 6-axis force-torque sensor and finger-tip pressure distribution sensors provided force control feedback. The robot manipulated pieces in a standard chess set, and a Mesa SwissRanger 4000 mounted overhead allowed it to locate the individual pieces. Domain-specific planning of chess moves was done with the Crafty chess engine [31]. The perception, motion planning, and control software was implemented primarily in C/C++ and Common Lisp using message-passing IPC [12] via shared memory and TCP running on Ubuntu Linux 10.04. The lowest-levels of our grammatical controller operate at a 1kHz rate.

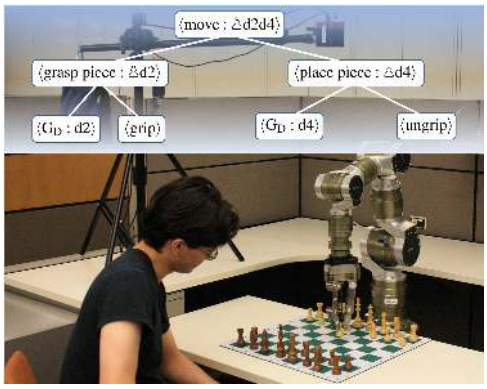


Fig. 4. Our experimental setup for human-robot chess and a partial parse-tree indicating the robot's plan to perform a chess move.

B. Tokenization

Tokens are the terminal symbols of the language, which we use to model discrete elements of the system. Tokens may be produced either *synchronously* or *asynchronously*. Synchronous tokens can represent a purely discrete *predicate*. For example, there is a token to indicate a winning position on the chessboard. Asynchronous tokens can represent entering a *region* within the continuous state space. These may be regions in which the underlying dynamics of the system change, for example a position where contact is made with another object. They may also be regions where we want our input to the system to abruptly change, for example a mobile robot reaching a waypoint and switching to a different trajectory. A new token is then generated when the robot enters into that region. This way, we only need a number of tokens equal to the number of events that cause a discrete change in the system. Such a minimalist approach avoids the exponential number of states produced by a grid-like discretization of high-dimensional spaces.

The tokens in our example Motion Grammar for chess are based on both the sensor readings and chessboard state. A summary of token types is given in Table I. Regions of interest are identified based on different thresholds. Position thresholds, velocity thresholds, and timeouts indicate when the robot has reached the end of a trajectory. Force thresholds and position thresholds indicate when the robot is in a safe operating range.

TABLE I
CHESS GRAMMAR TOKENS

Sensor Tokens		
Token	$\eta(z)$	**Description**
$[t_a < t \leq t_b]$	$t_a < t \leq t_b$	Trajectory Region
[limit]	$\|\mathbf{F}\| > F_{\max}$	Force Limit
[grasped]	$\int \rho dA > \varepsilon_{\int \rho}$	Pressure sum limit
[ungrasped]	$\neg$ [grasped]	Pressure sum limit
Perception Tokens		
Token	$\eta(z)$	**Description**
[obstacle]	$w(\mathbf{C}) < w_k$	Robot workspace occupied
[occupied(x)]	$w(x) > w_m in$	Piece is present in x
[clear(x)]	$\neg$ [occupied(x)]	No piece in x
[fallen(x)]	height$(x) < h_{\min}$	Piece is fallen
[offset(x)]	mean$(x) -$ pos$(x) > \varepsilon$	Piece is not centered
[moved]	$C_r \neq C_c$	Boardstate is different
[misplaced(x)]	$C_r(x) \neq C_c(x)$	Piece is missing
Chessboard Tokens		
Token		**Description**
[set]		board is properly set
[moved]		opponent has completed move
[checkmate]		checkmate on board
[resign]		a player has resigned
[draw]		players have agreed to draw
[cycle(x)]		x is in a cycle of misplaced pieces

We can define general regions via *level sets* $\mathscr{M}$, where $\mathscr{M} = \{x \,:\, s(x) = 0\}$ for scalar function $s(x)$. Then when the system crosses this boundary $\mathscr{M}$ for some region ζ, the tokenizer η generates ζ and passes it to the parser which expands the appropriate productions of the grammar.

C. Parsing

The *Motion Parser* reads in tokens and chooses the appropriate production from the grammar to expand and execute. This parser is derived from the Motion Grammar. Note that while the Context-Free model specifies an infinite-depth stack, physical computers are limited by available memory. This will restrict the maximum depth of the parse tree, though not the size of the input [1, p.226]. For our proof-of-concept application, we used a hand-written *recursive descent parser*, an approach also employed by GCC [22]. A recursive descent parser is written as a set of mutually-recursive procedures, one for each nonterminal in the grammar. An example of one of these procedures is shown in Algorithm 1, based on [1, p.219]. Each procedure will fully expand its nonterminals via a top-down, left-to-right derivation. This approach is a good match for the Motion Grammar's top-down task decomposition and its left-to-right temporal progression. In addition, there are a variety of algorithms for translation of grammars into parsers [1, 39] which may also be applied to Motion Grammars.

D. Syntax and Semantics

The syntax of the Motion Grammar represents the discrete system dynamics while the *semantic rules* in the grammar compute the continuous dynamics and control inputs. Within the Motion Parser, semantic rules are procedures that are executed when the parser expands a production. For our application, these rules store updated sensor readings, determine new targets for the controller, and send control inputs. These values are stored in the attributes of tokens and nonterminals.

Algorithm 1: parse-recursive-descent-A

```
Choose a production for A, A → X_1 ... X_n;
for i = 1 ... n do
    if nonterminal? X_i then
        call X_i;
    else if X_i = η(z(t)) then
        continue;
    else
        syntax error
Execute semantic rule for A → X_1 ... X_n;
```

PRODUCTION	SEMANTIC RULES
$\langle \mathrm{T} \rangle \rightarrow \langle \mathrm{T_1} \rangle \langle \mathrm{T_2} \rangle$	
$\langle \mathrm{T_1} \rangle \rightarrow \langle \mathrm{A_1} \rangle \langle \mathrm{A_2} \rangle$	
$\langle \mathrm{T_2} \rangle \rightarrow \langle \mathrm{A_3} \rangle \langle \mathrm{A_4} \rangle$	
$\langle \mathrm{A_1} \rangle \rightarrow [0 \leq \mathrm{t} < \mathrm{t_1}]$	$x_r = x_0 + \frac{1}{2}\ddot{x}_m t^2, \dot{x}_r = t\ddot{x}_m$
$\langle \mathrm{A_2} \rangle \rightarrow [\mathrm{t_1} \leq \mathrm{t} < \mathrm{t_2}]$	$x_r = x_0 + \frac{1}{2}\ddot{x}_m t_1^2 + \dot{x}_m(t - t_1), \dot{x}_r = \dot{x}_m$
$\langle \mathrm{A_3} \rangle \rightarrow [\mathrm{t_2} \leq \mathrm{t} < \mathrm{t_3}]$	$x_r = x_n - \frac{1}{2}\ddot{x}_m (t_3 - t)^2, \dot{x}_r = \dot{x}_m + \ddot{x}_m (t_2 - t)$
$\langle \mathrm{A_4} \rangle \rightarrow [\mathrm{t_3} \leq \mathrm{t}]$	$u = 0$

Fig. 5. Syntax-Directed Definition that encodes impedance control over trapezoidal velocity profiles. For each A_i, the input is computed according to $u = \dot{x}_r - K_p(x - x_r) - K_f(f - f_r)$.

Attributes for a nonterminal node in the parse tree are *synthesized* from child nodes and *inherited* from both the parent nodes and the left-siblings of that nonterminal. Here, we give a key example of robot control through semantic rules.

1) Example SDD: The Syntax-Directed Definition presented in Fig. 5 illustrates a simple grammar for implementing trapezoidal velocity profiles. Expanding $\langle \mathrm{A_i} \rangle$ will carry the system through the phases of the trajectory. While $[0 \leq \mathrm{t} < \mathrm{t_1}]$, the system will constantly accelerate according to $\langle \mathrm{A_1} \rangle$. While $[\mathrm{t_1} \leq \mathrm{t} < \mathrm{t_2}]$, the system will move with constant velocity according to $\langle \mathrm{A_2} \rangle$. While $[\mathrm{t_2} \leq \mathrm{t} < \mathrm{t_3}]$, the system will constantly decelerate according to $\langle \mathrm{A_3} \rangle$. Finally, the system will stop according to $\langle \mathrm{A_4} \rangle$. Each segment of the piecewise smooth trajectory is given by the semantic rule of one of the productions. This is an example of how the continuous domain control of physical systems can be encoded in the semantics of a discrete grammar.

2) Ordering of Syntax and Semantics: The online execution of the Motion Grammar also imposes constraints on the ordering of tokens and semantic rules. First, to move between two regions, represented as tokens, there must be some semantic rule to define this transition. Second, we cannot have two semantic rules without some other token to transition between them. Third, we need to define the continuous-domain initial conditions with some region token before any semantic rules. We can express these constraints *linguistically* by reconsidering the language of the Motion Grammar $\mathfrak{L}\{\mathscr{G}_M{}^\dagger\}$ as having three kinds of tokens: region tokens r, semantic rule tokens k, and other tokens p. That is, to produce $\mathscr{G}_M{}^\dagger$, we translate the

productions of $\mathscr{G}_M$ as follows,

$$P_{ij}^{\dagger} = \begin{cases} k & P_{ij} \in K \\ r & P_{ij} \in Z, \text{ region token} \\ p & P_{ij} \in Z, \text{ non-region token} \\ V_i^{\dagger} & P_{ij} \in V,\ P_{ij} = V_i \end{cases} \quad (3)$$

where $V_i, V_i^{\dagger}$ are the i^{th} nonterminals of $\mathscr{G}_M$ and ${\mathscr{G}_M}^{\dagger}$ and $P_{ij}, P_{ij}^{\dagger}$ are the j^{th} elements of the i^{th} productions of $\mathscr{G}_M$ and ${\mathscr{G}_M}^{\dagger}$. Then, we compare ${\mathscr{G}_M}^{\dagger}$ to the ordering constraints expressed as the intersection of the following regular expressions,

$$\mathfrak{L}\left\{{\mathscr{G}_M}^{\dagger}\right\} \subseteq \overline{\mathfrak{L}\{.^* r (\neg k)^* r.^*\}} \cap \overline{\mathfrak{L}\{.^* kk.^*\}} \cap \mathfrak{L}\{(\neg k)^* (r.^*)?\} \quad (4)$$

E. Completeness

For a robot to be reliable, it must respond to any feasible situation. This requires a *policy*. For a Motion Grammar model $\mathscr{G}_M$ of system F to represent a policy, it must include the set of all *paths* that the system can take. This property is given by the simulation $F \preceq \mathscr{G}_M$, "$\mathscr{G}_M$ simulates F." The concrete definition of a path depends on type of system we are dealing with. For discrete systems, a path is the sequence of states and transitions the system takes. For continuous systems, a path is the trajectory though its state space [24]. For the hybrid systems we consider here, paths and simulation have both continuous and discrete components. Using Def. 7 for path Ψ, we define simulation as follows,

Definition 13: Given $\mathscr{G}_M$ and system F with $x(t), x'(t), u(t), u'(t) \in \mathscr{X}_F, \mathscr{X}_{\mathscr{G}_M}, \mathscr{U}_F, \mathscr{U}_{\mathscr{G}_M}$ for time t and initial conditions $x_0, x_0' \in \mathscr{X}_F, \mathscr{X}_{\mathscr{G}_M}$. Then $F \preceq_c \mathscr{G}_M \equiv (x_0 = x_0{}' \wedge u(t) = u'(t) \implies x(t) = x'(t))$.

Definition 14: Given $\mathscr{G}_M$ and system F then $F \preceq_d \mathscr{G}_M \equiv \mathfrak{L}(F) \subseteq \mathfrak{L}(\mathscr{G}_M)$

Relation $F \preceq_c \mathscr{G}_M$ shows that F and $\mathscr{G}_M$ follow the same continuous trajectories. We match these trajectories exactly because a Motion Grammar must represent a policy and have LL(1) semantics – at each point along the path, $\mathscr{G}_M$ must specify a unique input u. Thus, Def. 13 precludes grammars which specify infeasible trajectories of the physical system, such as moving to unreachable configurations, because such a grammar would not contain the true system trajectory. When the system F's $x(t)$ does not match the grammar $\mathscr{G}_M$'s $x(t)$ for the specified input u, this does not satisfy $\preceq_c$.

Relation $F \preceq_d \mathscr{G}_M$ shows that the language of the system is a subset of the language of Motion Grammar. Note that for events which represent region entry, $F \preceq_d \mathscr{G}_M$ is implied by $F \preceq_c \mathscr{G}_M$. We define $\preceq_d$ separately in order to model some events as purely discrete with no continuous-domain component.

Definition 15: Given $\mathscr{G}_M$ and system F then $\mathsf{complete}\{\mathscr{G}_M\} \equiv F \preceq \mathscr{G}_M \equiv F \preceq_c \mathscr{G}_M \wedge F \preceq_d \mathscr{G}_M$

Relation $F \preceq \mathscr{G}_M$ means that $\mathscr{G}_M$ is a faithful model of F which captures relevant system behavior, that all feasible paths are represented by $\mathscr{G}_M$. Proving simulation between arbitrary systems is a difficult problem. In the purely discrete Context-Free case, it is undecidable [28, p.203]. However, we can always disprove completeness with a counterexample: for x and y, a path of x not defined by y would prove $x \not\preceq y$. Our main concern, though, is not simulation between any two systems but that our model $\mathscr{G}_M$ simulate the physical system we wish to control. In this work, we approach simulation and completeness as a modeling problem. We match the productions of the model $\mathscr{G}_M$ to the operating modes and events of F, though we do have the freedom to specify input u and define new regions or switching points as is convenient. For our chess application (Sect. VI), we manually designed the grammar based on the robot arm dynamics, the rules of chess game-play, and the interactions with the human. At this time, proving completeness or probabilistic completeness for general system models remains the subject of future work. However, in ongoing work, we are exploring some methods to automate construction of Motion Grammars [7, 9, 11].

When the system can be hierarchically decomposed, modeling events with a CFG provides a more compact representation than finite state models due the ability to reuse some productions in the CFG which would otherwise be duplicated in finite state models (e.g. sect. VI-B). However, naïve grid-based discretization of continuous spaces will produce a number of region tokens exponential in the number of dimensions (sect. V-B). In our sample implementation, we avoid this issue by considering region tokens only for the destination of a trajectory (sect. V-D).

In addition to providing a policy for the robot, a complete Motion Grammar has another important use: the grammar serves as an abstraction for the entire system. We can use this abstraction to prove that the modeled system is *correct*.

F. Correctness

Given a policy for the robot, it is crucial to evaluate the correctness of that policy. We define the correctness of a language specified as a Motion Grammar, $\mathfrak{L}(\mathscr{G}_M)$, by relating it to a *constraint language*, $\mathfrak{L}_r$. While $\mathfrak{L}(\mathscr{G}_M)$ for a given problem integrates all problem subtasks, as shown in Sect. VI, the constraint language targets correctness with respect to a specific criterion. Criteria can be formulated for general tasks, including safe operation, target acquisition, and the maintenance of desirable system attributes. By judiciously choosing the complexity of these languages, we can evaluate whether or not all strings generated by our model $\mathscr{G}_M$ are also part of language $\mathfrak{L}_r$.

Definition 16: A Motion Grammar $\mathscr{G}_M$ is correct with respect to some constraint language $\mathfrak{L}_r$ when all strings in the language of $\mathscr{G}_M$ are also in $\mathfrak{L}_r$: $\mathsf{correct}\{\mathscr{G}_M, \mathfrak{L}_r\} \equiv \mathfrak{L}(\mathscr{G}_M) \subseteq \mathfrak{L}_r$.

This approach to verifying correctness provides a model-based guarantee on behavior, ensuring proper operation of the discrete abstraction represented by $\mathscr{G}_M$. This verification of the *model* $\mathscr{G}_M$ ensures correctness of the underlying physical system F to the extent that $\mathscr{G}_M$ is *complete*, Def. 15. If we suppose system F contains some hybrid path ψ_{bad} with discrete component σ_{bad} and that ψ_{bad} is not in $\mathscr{G}_M$ – that is, $\mathscr{G}_M$ is not complete – then checking $\mathfrak{L}(\mathscr{G}_M) \subseteq \mathfrak{L}_r$ gives no information about whether $\sigma_{\text{bad}} \in \mathfrak{L}_r$. On the other hand, when $\mathscr{G}_M$ does contain the set of all feasible system paths, verifying $\mathscr{G}_M \subseteq \mathfrak{L}_r$ ensures correctness of all these paths. Thus, a complete model is necessary in order to meaningfully verify correctness.

The question of $\mathsf{correct}\{\mathscr{G}_M, \mathfrak{L}_\mathsf{r}\}$ is only decidable for certain language classes of $\mathfrak{L}(\mathscr{G}_M)$ and $\mathfrak{L}_r$. Hence, the formal guarantee on correctness is restricted to a limited range of complexity for both systems and constraints. We show decidability and undecidability for combinations of Regular, Deterministic Context-Free, and Context-Free Languages.

Lemma 17: Let $\mathscr{L}_R$, $\mathscr{L}_D$, and $\mathscr{L}_C$ be the Regular, Deterministic Context-Free, and Context-Free sets, respectively, and let $R \in \mathscr{L}_R$, $D, D' \in \mathscr{L}_D$, and $C, C' \in \mathscr{L}_C$. Then,

1) $C \subseteq C'$ is undecidable. [28, p.203]
2) $R \subseteq C$ is undecidable. [28, p.203]
3) $C \subseteq R$ is decidable. [28, p.204]
4) $R \subseteq D$ is decidable. [28, p.246]
5) $D \subseteq D'$ is undecidable. [28, p.247]

Corollary 18: Based on $\mathscr{L}_R \subset \mathscr{L}_D \subset \mathscr{L}_C$, the results from [28] extend to the following statements on decidability:

1) $D \subseteq R$ and $R \subseteq R$ are decidable.
2) $D \subseteq C$ is undecidable.
3) $C \subseteq D$ is undecidable.

Combining these facts about language classes, the system designer can determine which types of languages can be used to define both the grammars for specific problems and general constraints.

Theorem 19: The decidability of $\mathsf{correct}\{\mathscr{G}_M, \mathfrak{L}_\mathsf{r}\}$ for Regular, Deterministic Context-Free, and Context-Free Languages is specified by Fig. 6.

	$\mathfrak{L}_r \in \mathscr{L}_R$	$\mathfrak{L}_r \in \mathscr{L}_D$	$\mathfrak{L}_r \in \mathscr{L}_C$
$\mathfrak{L}(\mathscr{G}_M) \in \mathscr{L}_R$	yes	yes	no
$\mathfrak{L}(\mathscr{G}_M) \in \mathscr{L}_D$	yes	no	no
$\mathfrak{L}(\mathscr{G}_M) \in \mathscr{L}_C$	yes	no	no

Fig. 6. Decidability of $\mathsf{correct}\{\mathscr{G}_M, \mathfrak{L}_\mathsf{r}\}$ by language class.

Proof: Each entry in Fig. 6 combines a result from Lemma 17 or Corollary 18 with Definition 16. ■

Theorem 19 ensures that we can prove the correctness of a Motion Grammar with regard to any constraint languages in the permitted classes. We are limited to Regular constraint languages except in the case of a Regular system language which allows a Deterministic Context-Free constraint. Regular constraint languages may be specified as Finite Automata, Regular Grammars, or Regular Expressions since all are equivalent. We can also use Linear Temporal Logic as described in sect. VII-E.

To evaluate $\mathsf{correct}\{\mathscr{G}_M, \mathfrak{L}_\mathsf{r}\}$, consider $\mathfrak{L}(\mathscr{G}_M) \subseteq \mathfrak{L}_r$ as, "Does $\mathfrak{L}(\mathscr{G}_M)$ contain any string *not* in $\mathfrak{L}_r$?" which gives equation (5) [3, p.163].

We can explicitly evaluate (5) by computing the Regular $\overline{\mathfrak{L}_r}$ [28, p.59], intersecting this with $\mathfrak{L}(\mathscr{G}_M)$ [28, p.135], then testing the Context-Free result for emptiness [19]. These algorithms are implemented in the Motion Grammar Kit.

$$\mathfrak{L}(\mathscr{G}_M) \cap \overline{\mathfrak{L}_r} \stackrel{?}{=} \emptyset \quad (5)$$

G. Uncertainty

Robotic systems contain many sources of uncertainty. Linguistic approaches such as the Motion Grammar are well suited for addressing *unpredictable events* within the discrete dynamics. This occurs when at some point in time, the next token or discrete event is unknown. Other common sources of uncertainty include sensor noise, model error, and classification error.

$$\begin{aligned}
\langle \mathrm{G} \rangle &\rightarrow \langle \mathrm{T} \rangle \mid \langle \mathrm{L}_1 \rangle \\
\langle \mathrm{L}_1 \rangle &\rightarrow [0 < \mathrm{t} \leq \mathrm{t}_1]\langle \mathrm{L}_2 \rangle \mid [0 < \mathrm{t} \leq \mathrm{t}_1][\mathrm{limit}] \\
\langle \mathrm{L}_2 \rangle &\rightarrow [\mathrm{t}_1 < \mathrm{t} \leq \mathrm{t}_2]\langle \mathrm{L}_3 \rangle \mid [\mathrm{t}_1 < \mathrm{t} \leq \mathrm{t}_2][\mathrm{limit}] \\
\langle \mathrm{L}_3 \rangle &\rightarrow [\mathrm{t}_2 < \mathrm{t} \leq \mathrm{t}_3][\mathrm{limit}]
\end{aligned}$$

Fig. 7. Grammar fragment for guarded moves. $\langle \mathrm{T} \rangle$ is defined in Fig. 5

A *complete* Motion Grammar (Def. 15) addresses unpredictable events by representing a linguistic *policy* over all feasible events. For example, in the human-robot chess match, the robot safely responds to the uncertain event of the human entering the workspace (sect. VI-A). Such a complete grammar defines a language which contains all strings of events which may occur, thus representing a policy to respond to those events.

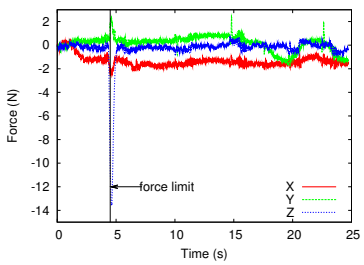
Uncertainty due to sensor noise was an issue present in our human-robot chess implementation. To address this, we incorporated a Kalman Filter into the semantic rules K. This effectively attenuated the noise due to electromagnetic interference for the strain gauges in the wrist force-torque sensor. While Kalman Filters often operate well in practice, they do not guarantee robustness [15]. Additionally, error in state estimation may result in an event triggering due to estimated state which would not trigger due to actual state. When this is possible, additional grammar productions to handle the erroneous triggering are necessary. Thus, while our implementation was tolerant of the noise present in the system, further work is needed to formally address sensor noise.

One issue which we do not currently address in the Motion Grammar is multiple hypothesis state estimation such as that performed by a particle filter. This is important for applications such as visual tracking of humans. Extensions to the Motion Grammar such as stochastic or parallel parsing could address multiple hypothesis estimation. In addition, one could also preprocess the sensor data, though this will exist outside of the guaranteed model that the Motion Grammar provides. This type of uncertainty requiring multiple hypothesis estimation remains as another area for improvement.

VI. Human-Robot Game Application

A. Guarded Moves

Our implementation of guarded moves using the Motion Grammar allows the human and robot to safely operate in the same workspace. A [limit] token is generated when the wrist force-torque sensor encounters forces above a preset limit. The limit is large enough so that the robot can perform its task and small enough to not injure the human or damage itself. When the parser detects [limit], it stops and backs off, preventing damage or injury. The plot in 8(a) shows the forces encountered by the robot in this situation. The large spike at 4.7s occurs when the robot's end-effector makes contact with



(a) Forces

(b) Contact

Fig. 8. Grammatical guarded moves safely protecting the human player.

the human's hand pictured in 8(b). The grammar in Fig. 7 *guarantees* that when this situation occurs, the robot will stop. After the human removes his hand from the piece, the robot can then safely reattempt its move

This example shows the importance of both response to uncertain events – the human entering the workspace – and fast online control possible with the Motion Grammar. The robot must respond immediately to the dangerous situation of impact with the human. The polynomial runtime performance of Context-Free parsers means that the grammatical controller can respond quickly enough, and the syntax of Fig. 7 guarantees that the robot will stop moving according to the kinematic model. For guarded moves with a dynamic model, the method from [13] could be incorporated in place of the kinematic model here.

1) Guarded Move Verification: We use a regular expression to verify the guarded move grammar fragment from Fig. 7, showing that the system will not continue after a force limit. This can be defined as,

$$\mathfrak{L}_{\mathscr{G}} \subseteq \mathfrak{L}\left\{(\neg[\text{limit}])^* [\text{limit}]?\right\} \tag{6}$$

The regular expression is equivalent to the FA in 9(a), where we see some arbitrary number of tokens that are not [limit] followed optionally by at most one [limit].

Claim 20: The grammar fragment in Fig. 7, $\mathscr{G}$, is correct with respect to (6).

Proof: We apply (5) to mechanically perform the verification. Each step is shown in Fig. 9. Since $\mathfrak{L}(\mathscr{G}_M) \cap \overline{\mathfrak{L}_r}$ is empty (no accept states in 9(d)), $\mathfrak{L}(\mathscr{G}_M) \subseteq \mathfrak{L}_r$. ■

B. Fallen Pieces

The grammar to set fallen pieces upright has a fairly simple structure but builds upon the previous grammars to perform a more complicated task, demonstrating the advantages of a hierarchical decomposition for manipulation. This grammar is shown in Fig. 10, and Fig. 11 shows a plot of the finger tip forces and pictures for this process. The production $\langle \text{recover} : \mathbf{x}, z \rangle$ will pick up fallen piece z at location $\mathbf{x}$. The nonterminal $\langle \text{T} : \mathbf{x} \rangle$ moves the arm to location $\mathbf{x}$. The production $\langle \text{pinch} \rangle$ will grasp the piece by squeezing tighter until the fingertip pressure sensors indicate a sufficient force. The production $\langle \text{T} : \mathbf{x} + \text{h}(z)\hat{k}, \frac{\pi}{6} \rangle$ will lift the piece sufficiently high above the ground and rotate it so that it can be replaced upright. Finally the nonterminal $\langle \text{release} \rangle$ will release the grasp on the piece setting it upright.

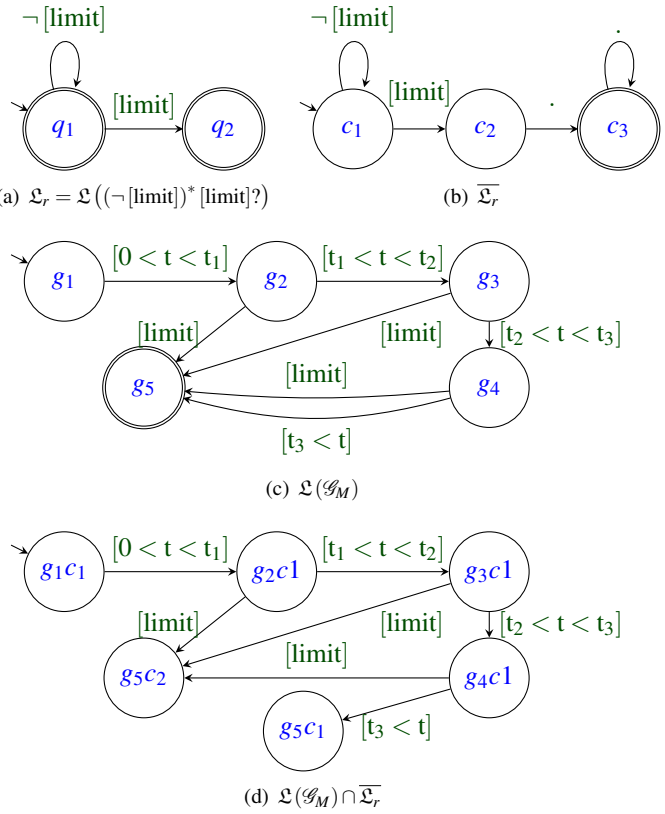
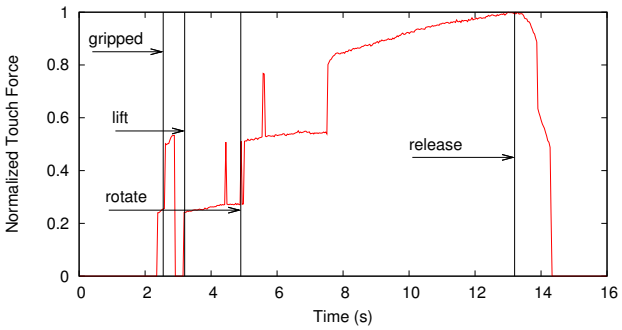


Fig. 9. Verification of Claim 20. Robot stops after single [limit] token.

$$\begin{aligned} \langle \text{recover} : \mathbf{x}, z \rangle &\rightarrow \langle \text{T} : \mathbf{x} \rangle \langle \text{pinch} \rangle \langle \text{T} : \mathbf{x} + \text{h}(z)\hat{k}, \tfrac{\pi}{6} \rangle \langle \text{release} \rangle \\ \langle \text{pinch} \rangle &\rightarrow [\text{grasped}] \mid [\text{ungrasped}] \langle \text{pinch} \rangle \end{aligned}$$

Fig. 10. Grammar fragment for recovering fallen pieces



(a) Touch Force: Knight

(b) Grasped, Rook (c) Rotated, Queen (d) Finished, Bishop

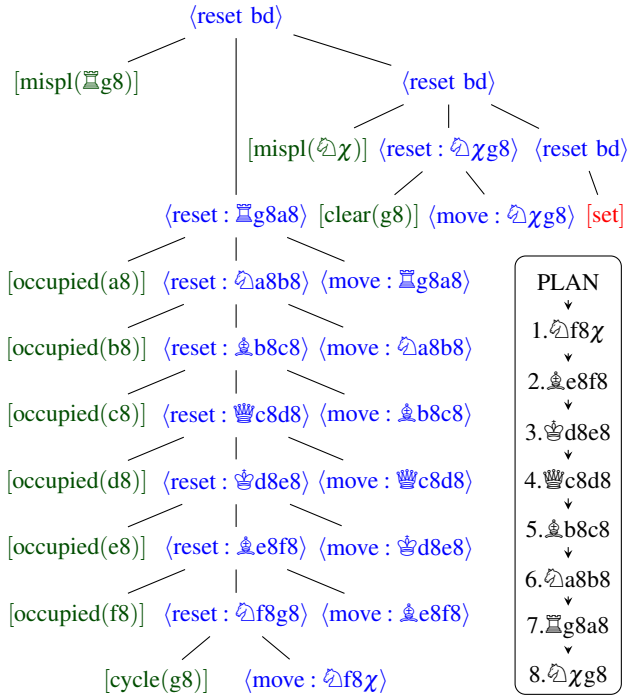
Fig. 11. Robot recovering fallen pieces

$$
\begin{array}{lcl}
\langle \text{reset board} \rangle & \rightarrow & [\text{set}] \,|\, [\text{misplaced}(x)] \langle \text{reset} : x, \text{home}(x) \rangle \langle \text{reset board} \rangle \\
\langle \text{reset} : x_0, x_1 \rangle & \rightarrow & [\text{clear}(x_1)] \langle \text{move} : x_0, x_1 \rangle \\
& | & [\text{occupied}(x_1)] \langle \text{reset} : x_1, \text{home}(x_1) \rangle \langle \text{move} : x_0, x_1 \rangle \\
& | & [\text{cycle}(x_1)] \langle \text{move} : x_1, \text{rand}() \rangle
\end{array}
$$

Fig. 12. Grammar fragment to reset chessboard

(a) Board position - Initial

(b) Board position - Final



(c) Motion grammar parse tree and plan for resetting the board.

Fig. 13. Example of board resetting

C. Board Resetting

The problem of resetting the chess board presents an interesting grammatical structure. If the home square of some piece is occupied, that square must first be cleared before the piece can be reset. Additionally, if a cycle is discovered among the home squares of several pieces, the cycle must be broken before any piece can be properly placed. The grammatical productions to perform these actions are given in Fig. 12.

An example of this problem is shown in 13(a) where all of Blacks's Row 8 pieces have been shifted right by one square. The parse tree for this example is shown in 13(c), rooted at ⟨reset board⟩. As the robot recurses through the grammar in Fig. 12, chaining an additional ⟨reset⟩ for each occupied cell, it eventually discovers that a cycle exists between the pieces to move. To break the cycle, one piece, ♘f8, is moved to a random free square, χ. With the cycle broken, all the other pieces can be moved to their home squares. Finally, ♘χ can be moved back to its home square. This sequence of board state tokens and ⟨move⟩ actions can be seen by tracing the leaves of the parse tree as shown beginning from PLAN in 13(c).

Observe that as the parser searches through the chain of pieces that occupy each other's home squares, it is effectively building up a stack of the moves to make. This demonstrates the benefits of the increased power of Context Free Languages over the Regular languages commonly used in other hybrid control systems. Regular languages, equivalent to finite state machines, lack the power to represent this arbitrary depth search.

Claim 21: Let n be the number of misplaced pieces on the board. The grammar in Fig. 12 will reset the board with at most $1.5n$ moves.

Proof: Every misplaced piece not in a cycle takes one move to reset to its proper square. Every cycle causes one additional move in order to break the cycle. A cycle requires two or more pieces, so there can be at most $0.5n$ cycles. Thus one move for every piece and one move for $0.5n$ cycles give a maximum of $1.5n$ moves. ■

1) Board Resetting Verification: We use a Linear Temporal Logic (LTL) formula to verify the board resetting grammar fragment from Fig. 12, showing that eventually, the board will be set. This can be defined as,

$$\mathfrak{L}_{\mathscr{G}} \subseteq \mathfrak{L}(\Diamond [\text{set}]) \qquad (7)$$

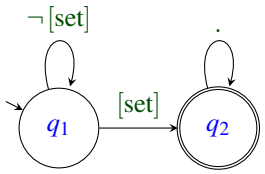


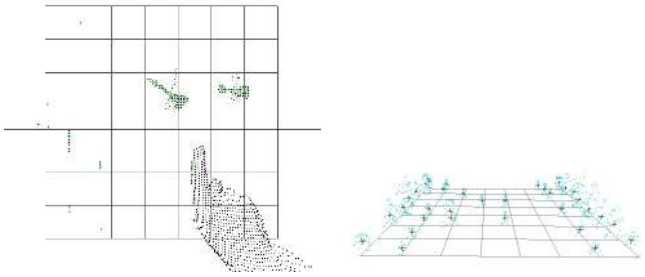
Fig. 14. Automata for Correctness Specification $\Diamond$[set].

The LTL formula is equivalent to the automaton in Fig. 14, where we see that the token [set] must at some point occur.

Claim 22: The grammar fragment in Fig. 12, $\mathscr{G}$, is correct with respect to (7).

Proof: The mechanical verification uses (5) and follows the proof of Claim 20. First, we convert Fig. 12 to Pushdown Automaton P and specification $\Diamond$[set] to Büchi Automaton S. Then, we compute $\mathfrak{L}(P) \cap \overline{\mathfrak{L}(S)}$. The result is the empty set, so the specification is satisfied. ■

Note that there is one potential caveat with the guarantees of LTL formulas of the form $\Diamond x$. When this formula is satisfied, it is allowable to have an arbitrary number of $\neg x$ tokens before any x is seen. A similar issue exists for the Kleene Closure (*) operator in Regular Expressions. Consider the LTL formula and equivalent Büchi automaton is Fig. 14 to see how $\Diamond$ corresponds to automaton state transitions. Informally stated, $\Diamond x$ and $(\neg x)^* x$ both mean that we will see an arbitrary number of $\neg x$, but we will keep getting tokens until we do get that x. If a specific finite limit of $\neg x$ is desired, then this must either be explicitly stated or addressed through *fairness* [3, p.126] assumptions eliminating unrealistic infinite behavior.



(a) Detecting fallen pieces. (b) Finding offsets for all pieces

Fig. 15. Perception with point cloud is discretized into tokens.

$$
\begin{array}{rcl}
\langle \text{game} \rangle & \rightarrow & \langle \text{act} \rangle \langle \text{end} \rangle \mid \langle \text{act} \rangle \langle \text{game}' \rangle \\
\langle \text{game}' \rangle & \rightarrow & \langle \text{wait} \rangle \langle \text{end} \rangle \mid \langle \text{wait} \rangle \langle \text{game} \rangle \\
\langle \text{end} \rangle & \rightarrow & [\text{checkmate}] \mid [\text{resign}] \mid [\text{draw}] \\
\langle \text{act} \rangle & \rightarrow & \langle \text{fix} \rangle \langle \text{turn} \rangle \langle \text{fix} \rangle \\
\langle \text{fix} \rangle & \rightarrow & \langle \text{end} \rangle \mid [\text{fallen} : \text{x}, \text{z}] \langle \text{recover} : \text{x}, \text{z} \rangle \langle \text{fix} \rangle \mid \varepsilon \\
\langle \text{turn} \rangle & \rightarrow & \langle \text{move} : \text{x}_0, \text{x}_1 \rangle \mid \langle \text{capture} : \text{x}_0, \text{x}_1 \rangle \\
 & \mid & \langle \text{castle} \rangle \mid \langle \text{castle queen} \rangle \mid \langle \text{en passant} \rangle \\
 & \mid & \langle \text{resign} \rangle \mid \langle \text{draw} \rangle \\
\langle \text{wait} \rangle & \rightarrow & [\text{moved}] \mid \langle \text{wait} \rangle \\
\langle \text{move} : \text{x}_0, \text{x}_1 \rangle & \rightarrow & \langle \text{grasp piece} : \text{x}_0 \rangle \langle \text{place piece} : \text{x}_1 \rangle \\
\langle \text{grasp piece} : \text{x} \rangle & \rightarrow & \langle \text{L} : \text{x} \rangle \langle \text{grasp piece} : \text{x} \rangle \mid \langle \text{T} : \text{x} \rangle \langle \text{grip} \rangle \\
\langle \text{place piece} : \text{x} \rangle & \rightarrow & \langle \text{L} : \text{x} \rangle \langle \text{place piece} : \text{x} \rangle \mid \langle \text{T} : \text{x} \rangle \langle \text{ungrip} \rangle \\
\langle \text{grip} \rangle & \rightarrow & [\text{grasped}] \mid [\text{ungrasped}] \langle \text{grip} \rangle \\
\langle \text{capture} : \text{x}_0, \text{x}_1 \rangle & \rightarrow & \langle \text{take} : \text{x}_1 \rangle \langle \text{move} : \text{x}_0, \text{x}_1 \rangle \\
\langle \text{take} : \text{x} \rangle & \rightarrow & \langle \text{move} : \text{x}, \text{offboard} \rangle \\
\langle \text{castle} \rangle & \rightarrow & \langle \text{move} : \text{♔e1g1} \rangle \langle \text{♖h1f1} \rangle \\
\langle \text{castle queen} \rangle & \rightarrow & \langle \text{move} : \text{♔e1c1} \rangle \langle \text{♖a1d1} \rangle \\
\langle \text{en passant} : \text{x} \rangle & \rightarrow & \langle \text{take} : \text{x} - 1 \rangle \langle \text{move} : \text{♙x} \rangle \\
\langle \text{resign} \rangle & \rightarrow & \langle \text{L} : \text{♔} + 1 \rangle \langle \text{resign} \rangle \mid \langle \text{T} : \text{♔} + 1 \rangle \langle \text{resign}' \rangle \\
\langle \text{resign}' \rangle & \rightarrow & \langle \text{L} : \text{♔} - 1 \rangle \langle \text{resign}' \rangle \mid \langle \text{T} : \text{♔} - 1 \rangle
\end{array}
$$

Fig. 16. Grammar Productions for Chess Game

D. Perception and Board Tokens

To play chess, we combined our grammatical controller with the Crafty [31] chess engine. The Crafty *boardstate* serves as the model of the position of the chessboard. The MESA SR4000 point cloud is tokenized into the perception symbols in Table I. To find the pieces, [obstacle], we cluster the point cloud, then weight each cluster $\mathbf{C}$ by the number of points in the cluster, $w(\mathbf{C})$. The height of each cluster is sufficient to classify an upright piece. For pieces that have fallen, we detect this case when the ratio of width and height exceeds a threshold and use the principal axis in the horizontal plane to find piece orientation. Fig. 15 shows these attributes in the point cloud. A nearest neighbor search over the entire chessboard determines all squares x with $[\text{occupied}(\text{x})]$. Piece offsets from square centers are computed and denoted by $[\text{offset}(\text{x})]$. The boardstate retrieved from perception C_r and the one from the Crafty engine C_c are compared to see whether a move has been made. If a move has been made, then $[\text{clear}(\text{x})]$ and $[\text{misplaced}(\text{x})]$ are determined. All of these tokens are input to the Motion Parser which then determines the next motion action for the chess game.

E. Full Game

The entire motion planning and control policy is specified in the grammar in Fig. 16. This grammar describes the game, $\langle \text{game} \rangle$, as consisting of an alternating sequence of the robot moving, $\langle \text{act} \rangle$, followed by the human moving, $\langle \text{wait} \rangle$, until the game has ended, $\langle \text{end} \rangle$, via checkmate, resignation, or draw. When it is the robot's turn, it will correct any fallen pieces, $\langle \text{fix} \rangle$, make its move, and then again correct any pieces that may have fallen while it was making the move. Making a move, $\langle \text{turn} \rangle$, can be either a simple move between squares, a capture, a castle, en passant, or a draw or resignation. A simple piece move, $\langle \text{move} \rangle$, requires first grasping the piece, then placing it on the correct square. To grasp the piece, the robot will move its hand around the piece then tighten its grip, $\langle \text{grip} \rangle$, until there is sufficient pressure registered on the touch sensors. To capture a piece, the robot will remove the captured piece from the board, $\langle \text{take} \rangle$, and then move the capturing piece onto that square. A $\langle \text{castle} \rangle$ requires the robot to move both the rook and the king. For $\langle \text{en passant} \rangle$, the robot will $\langle \text{take} \rangle$ the captured pawn and then move its own pawn to the destination square. Finally, to resign – indicating a failure in chess strategy, not motion planning – the robot moves its end-effector through the square occupied by the king, knocking it over. By following the rules of this grammar, our system will play chess with the human opponent.

VII. Relationship with Existing Methods

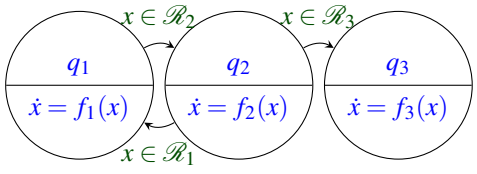
The Motion Grammar builds on a number of advances in linguistic control. This section relates our approach to several similar methods: Petri Nets, Hybrid Automata, MDLe, Maneuver Automata, Linear Temporal Logic, and the C Programming Language.

A. Petri Nets

Petri Nets are a modeling technique for discrete event systems based on a bipartite graph that represents the structure and dependencies of event firing. They are often used to model concurrent systems while CFGs generally represent a sequential structure. The languages that can be represented by a Petri Net are distinct from the Context-Free set. The language of some string followed by its reverse, $\left\{ww^R | w \in Z^*\right\}$, is Context-Free, but it is not a Petri Net language. The language of sequences of equal numbers of a, b, and c, $\{a^n b^n c^n\}$, is not Context-Free but can be represented by a Petri Net. However, the Petri Net languages are a strict superset of the Regular set and a strict subset of the Context-Sensitive set [40]. In consequence, the syntactic class of systems which can be modeled by a Petri Net is distinct from those modeled by the Context-Free Motion Grammar.

B. Hybrid Automata

Hybrid Automata represent a system with both event and time-driven dynamics. The system has a number of modes $q \in Q$. Each mode q_i is governed by some differential equation f_i. Transitions between modes occur in response to discrete events. The modes Q are generally finite [2, 26], so we can



(a) Hybrid Automaton

Production	Semantic Rules
$\langle Q_1 \rangle \rightarrow [x \in \mathscr{R}_2]\, Q_2$	$\dot{x} = f_1(x)$
$\langle Q_2 \rangle \rightarrow [x \in \mathscr{R}_1]\, Q_1$	$\dot{x} = f_2(x)$
$\mid [x \in \mathscr{R}_3]\, Q_3$	$\dot{x} = f_2(x)$
$\langle Q_3 \rangle \rightarrow \varepsilon$	$\dot{x} = f_3(x)$

(b) Motion Grammar

Fig. 17. Example of Hybrid Automata to Motion Grammar Conversion

represent these transitions with a Finite Automaton. Many descriptions of Hybrid Automata also define *jump sets* or *reset conditions* which discontinuously change state x; this is not a feature we consider in this analysis.

A Hybrid Automaton with finite control states or modes Q can be transformed into an equivalent Motion Grammar. This is possible because every Finite Automaton is equivalent to a Regular Grammar, and Regular Grammars are a subset of Context-Free Grammars. An example of this process for a three-state system is shown in Fig. 17. The algorithm to perform this transformation is given by Algorithm 2. Because the Motion Grammar is Context-Free, the reverse is not always possible, and there are Motion Grammars, such as Fig. 12, with no equivalent finite mode Hybrid Automaton.

Algorithm 2: HA-to-$\mathscr{G}_M(Q,\Sigma,E,F)$

Input: Q : set of discrete states
Input: Σ : alphabet of tokens
Input: E : set of edges, $Q \times Q$
Input: F : set of continuous dynamics functions associated with each state in Q

foreach $q_i \in Q$ **do**
  Create nonterminal $\langle Q_i \rangle$;
foreach $\sigma_i \in \Sigma$ **do**
  Create token $[\sigma_i]$;
foreach $e_j \in E,\ e_j : q_i \times \sigma_j \mapsto q_k$ **do**
  Create production $Q_i \rightarrow [\sigma_j]\, Q_k$ with semantic rule $\dot{x} = f_i(x)$;

C. MDLe

The MDLe is a Modeling Language with a Context-Free grammar [29]. Each string in the MDLe represents some control program. While the *modeling* (sect. IV-D) language MDLe is Context-Free, each of MDLe control programs can parse only a Regular Language *system* language. This is in contrast to the Motion Grammar which describes the *System Language* for a *Context-Free System*.

Theorem 23: The System Language recognized by an MDLe string is Regular.

Proof: Given that an MDLe controller is represented by a string in the MDLe language, we prove that the resulting System Language is regular by providing an algorithm to transform any MDLe string, Σ, into a Finite Automaton, $A = (S,E,d)$ that accepts the System Language $\mathfrak{L}_g$. MDLe string Σ is composed of tokens [(], [)], [,], controllers $u \in U$, and interrupts $\xi \in B'$. Algorithm 3 creates the automaton A corresponding to Σ. Notice that any u or ξ which appears multiple times in Σ results in multiple states in the FA.

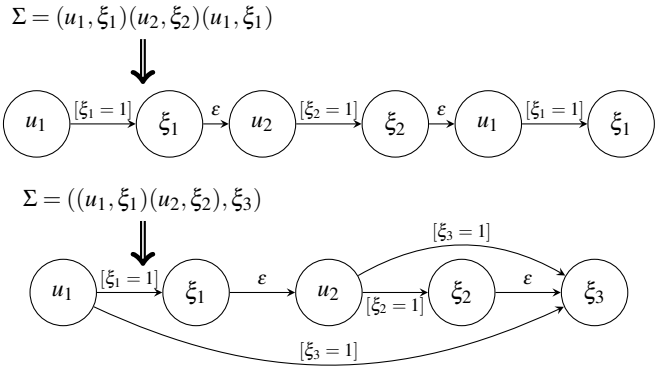


Fig. 18. Example Transform: MDLe to Finite Automata

The resulting Finite Automaton encodes the evaluation rules for the MDLe string. Since we can transform Σ to a Finite Automaton, Σ must recognize a Regular System Language. ■

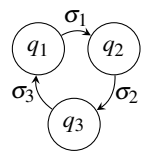
Algorithm 3: MDLe-to-FA(Σ,U,B')

Input: Σ : MDLe specification string
Input: U : set of controllers
Input: B' : set of interrupts
/* Create States */
$S = \Sigma - \{[(], [)], [,]\}$;
/* Create Transitions */
foreach $s \in S$ **do**
  if $s \in U$ **then**
    foreach ξ_i *enclosing* s *in* Σ **do**
      Create a transition $\left(s \xrightarrow{\xi_i=1} \xi_i\right)$;
  if $s \in B'$ **then**
    Create a transition $\left(s \xrightarrow{\varepsilon} r\right)$, where r is the next σ_i following s in Σ such that $r \in S$;

Two examples of this conversion procedure are shown in Fig. 18, one simple case and one more complicated case. Unlike the transformation to Hybrid Automata in [29], we do not restrict repeated controllers in Σ to a single state in our system language Finite Automata. Notice also that there is ambiguity in the case of simultaneously active interrupt functions. [29] specifies that this is resolved via precedence among the different interrupts.

Corollary 24: Every MDLe string can be translated to a Motion Grammar.

Proof: The Motion Grammar is a Context-Free grammar for the System Language, and we can translate every MDLe string to a Finite Automaton accepting the System Language. Finite Automata are equivalent to Regular Grammars. Regular



$$\begin{aligned}\langle q_1\rangle &\rightarrow [\sigma_1]\langle q_2\rangle\\ \langle q_2\rangle &\rightarrow [\sigma_2]\langle q_3\rangle\\ \langle q_3\rangle &\rightarrow [\sigma_3]\langle q_1\rangle\end{aligned}$$

(a) Offline Grammar

$$\begin{aligned}\langle q_1'\rangle &\rightarrow [x\in\mathscr{R}_1] \mid [x\in\mathscr{R}_1]\langle\sigma_1'\rangle\\ \langle\sigma_1'\rangle &\rightarrow \kappa_1\langle q_2'\rangle\\ \langle q_2'\rangle &\rightarrow [x\in\mathscr{R}_2] \mid [x\in\mathscr{R}_2]\langle\sigma_2'\rangle\\ \langle\sigma_2'\rangle &\rightarrow \kappa_2\langle q_3\rangle\\ \langle q_3'\rangle &\rightarrow [x\in\mathscr{R}_3] \mid [x\in\mathscr{R}_3]\langle\sigma_3'\rangle\\ \langle\sigma_3'\rangle &\rightarrow \kappa_3\langle q_1'\rangle\end{aligned}$$

(b) Online Grammar

Fig. 19. Maneuver Automaton → Online Grammar.

Grammars are a subset of Context-Free Grammars. ■

From Corollary 24, we also observe that the Motion Grammar can control a broader class of systems than the MDLe. MDLe controllers accept only Regular Languages while the Motion Grammar accepts Context-Free languages with LL(1) semantics, which include all Regular Languages. Thus, the Motion Grammar can describe systems that the MDLe cannot.

D. Maneuver Automata

There are some important similarities between the Maneuver Automaton and the Motion Grammar. The Maneuver Automaton represents a hybrid system moving between a set of *trim* trajectories $q \in Q$ using a motion library of *maneuvers* $\sigma \in \Sigma$ [20]. This system is represented as a Finite Automaton with states Q and tokens Σ. It is possible to transform this representation into a grammar suitable for online control of the system. An example of this process is shown in Fig. 19. First, the Maneuver Automaton, 19(a) is rewritten as a Regular Grammar, G_o in 19(a), with one production of the form $\langle q_i\rangle \rightarrow [\sigma_j]\langle q_k\rangle$ to indicate each transition in the automaton. We then transform this offline grammar into an online grammar G_n according to Algorithm 4. Entry into a trim state is marked by a region of the continuous state space $x \in \mathscr{R}$. The controller for some maneuver σ is given by a semantic rule κ_σ.

Algorithm 4: G_o-to-$G_n(G_o)$

```
/* Productions from states                          */
foreach ⟨q_i⟩ in G_o do
    Create production ⟨q_i'⟩ → [x ∈ R_{q_i}];
/* Productions from transitions                     */
foreach ⟨q_i⟩ → [σ_j]⟨q_k⟩ in G_o do
    Create production ⟨q_i'⟩ → [x ∈ R_{q_i}]⟨σ_j'⟩ ;
    Create production ⟨σ_j'⟩ → κ_{σ_j}⟨q_k'⟩;
```

We also note that an arbitrary Maneuver Automaton cannot be directly transformed into a Motion Grammar. The Maneuver Automaton does not include information about how long to hold in trim states q or when to begin maneuvers σ. Thus, it does not represent a policy and it can be transformed only to a grammar that is not Semantically LL(1). Thus, Claim 9 indicates that it cannot be a Motion Grammar.

Even though we cannot directly transform a Maneuver Automaton to a Motion Grammar, this transformation is possible by adding the additional information necessary for LL(1) Semantics, such as by establishing precedence levels between conflicting productions or extending the representation to include tokens such as timeouts for coasting times. By augmenting the Maneuver Automaton with the additional information to achieve a policy, we can then derive a corresponding Motion Grammar.

E. Linear Temporal Logic

Linear Temporal Logic (LTL) is an extension to propositional logic that describes the behavior of discrete systems over an infinite time horizon. This is an often convenient notation to specify various system properties. Every statement in LTL can be represented as a Büchi automaton; an example is Fig. 20. Büchi automata are a variation on Regular automata that describe infinite length strings [3]. We can restate classical automata over finite length strings as a special case of automata over infinite length strings by looping through the accept state of a classical automaton [27, p.131].

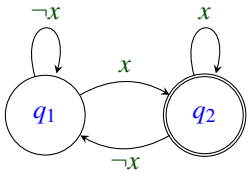


Fig. 20. Example of equivalence between Büchi Automata and Linear Temporal Logic formula $\square\lozenge x$.

Definition 25 (Stutter Extension): The stutter extension of finite string σ accepted by automaton A which halts with accept state q_n is the ω-run $\sigma,(q_n,\varepsilon,q_n)^\omega$ [27].
Alternatively, we can specify that some LTL property α holds only until a particular terminating condition, \$, by replacing all $\square\alpha$ with $\alpha\cup\$$. Because of the correspondence between LTL and formal language, we may also use LTL formulas to describe correctness of the Motion Grammar. One algorithm for checking Context-Free systems with LTL is given by [17].

F. The C Programming Language

The C programming language is a Turing-Complete computational model while the Motion Grammar is Context-Free. Rice's theorem means that for an arbitrary C program, *we can guarantee nothing*, not even that it halts! Because the Motion Grammar is restricted to Context-Free computation, the Earley parser [16] means online parsing will have worst case polynomial runtime. Furthermore, Theorem 19 means that for an arbitrary Motion Grammar, *we can always verify it* against an arbitrary Regular specification. This makes clear the trade-off we have made: sacrifice computational power to guarantee runtime performance and verifiability. As a practical matter, though, any Motion Grammar may be transformed into a C program since all Context-Free languages are Turing-Recognizable.

VIII. Conclusions and Future Work

In this paper we analyzed the discrete dynamics of hybrid systems from a Formal Language perspective. We presented a new system representation based on Context-Free Grammars which guarantees online computational efficiency and model-based verifiability. We analyzed the linguistic properties of

this Motion Grammar, showing the capabilities and limits of these formal guarantees and explained some particular constraints that arise in applying grammars to time-based physical systems. By relating several existing hybrid control techniques with the Motion Grammar, we showed the common linguistic representation these methods share. Finally, we have demonstrated the efficacy of this approach by developing a robotic system to play physical chess against a human opponent, showing both offline verification and computationally efficient online control.

Our software which implements this verification and parser generation approach is available at http://www.golems.org/node/1224.

This work presents many possibilities for automating the development and verification of controllers. In ongoing work, we are automating the construction of Motion Grammars [7, 9, 11]. There are also some possibilities for enhancing the power and guarantees of this method. Applying type theory could provide for stricter definitions and guarantees. There are restricted classes of Context-Sensitive languages that can be efficiently parsed if the Context-Free model for the Motion Grammar is insufficiently powerful for some problems [33]. We will continue exploring these approaches to improve the capabilities and guarantees of the resulting system.

Acknowledgments

The authors thank Magnus Egerstedt for his insight throughout the development of the Motion Grammar and Pushkar Kolhe for his work on the perception component of our implementation.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson, 2nd edition, 2007.

[2] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Hybrid Systems*, pages 209–229, 1993.

[3] C. Baier, J.P. Katoen, et al. *Principles of Model Checking*. MIT Press, Cambridge, MA, 2008.

[4] RW Brockett. Formal languages for motion description and map making. *Robotics*, 41:181–191, 1990.

[5] C.G. Cassandras and Stéphane Lafortune. *Introduction to Discrete-Event Systems*. Springer, 2nd edition, 2008.

[6] J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson, 3rd edition, 2005.

[7] N. Dantam, I. Essa, and M. Stilman. Linguistic transfer of human assembly tasks to robots. In *IROS*, 2012.

[8] N. Dantam, P. Kolhe, and M. Stilman. The motion grammar for physical human-robot games. In *ICRA*. IEEE, 2011.

[9] N. Dantam, C. Nieto-Granda, H. Christensen, and M. Stilman. Linguistic composition of semantic mapping and hybrid control. In *ISER*, 2012.

[10] N. Dantam and M. Stilman. The motion grammar: Linguistic planning and control. In *RSS*. IEEE, 2011.

[11] N. Dantam and M. Stilman. The motion grammar calculus for context-free hybrid systems. In *ACC*, 2012.

[12] N. Dantam and M. Stilman. Robustness and efficient communication for real-time multi-process robot software. In *Humanoids*. IEEE, 2012.

[13] A. De Luca, A. Albu-Schaffer, S. Haddadin, and G. Hirzinger. Collision detection and safe reaction with the DLR-III lightweight manipulator arm. In *IROS*, pages 1623–1630. IEEE/RSJ, 2006.

[14] A. De Santis, B. Siciliano, A. De Luca, and A. Bicchi. An atlas of physical human-robot interaction. *Mechanism and Machine Theory*, 43(3):253–270, 2008.

[15] John C. Doyle. Guaranteed margins for lqg regulators. *IEEE Trans. on Automatic Control*, 23(4):756–757, 1978.

[16] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[17] J. Esparza, A. Kučera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003.

[18] G.E. Fainekos, H. Kress-Gazit, and G.J. Pappas. Hybrid controllers for path planning: a temporal logic approach. In *CDC*. IEEE, 2005.

[19] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9:27–37, 1997.

[20] E. Frazzoli, MA Dahleh, and E. Feron. Maneuver-based motion planning for nonlinear systems with symmetries. *IEEE Trans. on Robotics*, 21(6):1077–1091, 2005.

[21] K. Fu. *Syntactic Pattern Recognition and Applications*. Prentice-Hall, 1981.

[22] July 2010. http://gcc.gnu.org/gcc-3.4/changes.html.

[23] M. Giuliani, C. Lenz, T. Müller, M. Rickert, and A. Knoll. Design principles for safety in human-robot interaction. *Intl. Journal of Social Robotics*, pages 1–22, 2010.

[24] E. Haghverdi, P. Tabuada, and G.J. Pappas. Bisimulation relations for dynamical, control, and hybrid systems. *Theoretical Computer Science*, 342(2-3):229–261, 2005.

[25] F. Han and S.C. Zhu. Bottom-up/top-down image parsing by attribute graph grammar. In *ICCV*, volume 2, 2005.

[26] T.A. Henzinger. The theory of hybrid automata. In *Symposium on Logic in Computer Science*, pages 278–292. IEEE, 1996.

[27] G.J. Holtzman. *The Spin Model Checker*. Addison Wesley, Boston, MA, 2004.

[28] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading, MA, 1979.

[29] D. Hristu-Varsakelis, M. Egerstedt, and PS Krishnaprasad. On the structural complexity of the motion description language mdle. In *CDC*, pages 3360–3365. IEEE, 2003.

[30] D. Hristu-Varsakelis and W.S. Levine, editors. *Handbook of Networked and Embedded Control Systems*. Birkhauser, 2005.

[31] RM Hyatt. CRAFTY–chess program. *ftp://ftp.cis.uab.edu/pub/hyatt*, 1996.

[32] L.T. Jones, A. Howden, M.S. Knighton, A. Sims, D.L. Kittinger, and R.E. Hollander. Robot computer chess game, Aug. 1983. US Patent 4,398,720.

[33] A.K. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of mildly context-sensitive grammar formalisms. *Foundational Issues in Natural Language Processing*, pages 31–81, 1991.

[34] M. Kloetzer and C. Belta. Automatic deployment of distributed teams of robots from temporal logic motion specifications. *IEEE Trans. on Robotics*, 26(1):48–61, 2010.

[35] P. Koutsourakis, L. Simon, O. Teboul, G. Tziritas, and N. Paragios. Single view reconstruction using shape grammars for urban environments. In *ICCV*, 2009.

[36] H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Trans. on Robotics*, 25(6):1370–1381, 2009.

[37] J. Lygeros, K.H. Johansson, S.N. Simic, J. Zhang, and S.S. Sastry. Dynamical properties of hybrid automata. *IEEE Trans. on Automatic Control*, 48(1):2–17, 2003.

[38] C. Matuszek, B. Mayton, R. Aimi, M.P. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J.R. Smith, and D. Fox. Gambit: An autonomous chess-playing robotic system. In *ICRA*, pages 4291–4297. IEEE, 2011.

[39] T. Parr and K. Fisher. LL (*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 425–436. ACM, 2011.

[40] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

[41] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *Analysis and Optimization of Systems*, 25(1):206–230, January 1987.

[42] Cheta Rawal, Herbert G. Tanner, and Jeffrey Heinz. (sub)regular robotic languages. In *IEEE Mediterranean Conference on Control and Automation*. IEEE, 2011.

[43] B. Stilman. *Linguistic Geometry: From Search to Construction*. Kluwer Academic Publishers, 2000.

[44] A. Toshev, P. Mordohai, and B. Taskar. Detecting and parsing architecture at city scale from range data. In *CVPR*. IEEE, 2010.

[45] D. Urting and Y. Berbers. Marineblue: A low-cost chess robot. *Robotics and Applications*, pages 76–81, June 2003.