Symbolic Constraint Language and Solver

Isaac Evans and Joseph Lynch

April 1, 2013

1 Introduction

We propose to built a system licensed under the MIT License for expressing and solving complex symbolic constraints between three-dimensional objects. This project would significantly expand on the concepts and applications presented in the Propagator, Pattern Matching, and Generic Operations aspects of the course.

A potential end use for this system is inclusion into the codebase for MIT's entry to the DARPA Robotics Challenge (DRC). One of our team members is currently a student researcher in Prof. Seth Teller's lab, which is directing MIT's competition entry. Our system will be built with robotic applications in mind, but our design is crafted in a way that is careful to avoid any architectural dependence on DRC code or conceptual dependency on the idea of constraints specific to a robot.

The term "symbolic constraint" implies constraints that are not simply fixed offsets. We envision this system being expressive enough to let users author constraints such as "these hands grasp the ladder rung at a point not so close to the edge that it cannot be reached, but not so close to the center that the hands are more than a shoulder-width apart." Such a constraint should be applicable not just to a single ladder instance, but also ladders of varying type–with cylindrical/square rungs, or even climbable objects in general such as stairs. The goal is that system users can specify behaviors that match their "high level conception" of how the constraint should follow in varying environments/objects.

In its simplest form, this is an extension from two to three dimensions of some high-level GUI frameworks, which offer "fluid" layouts an allow users to visualize the widget positioning for an arbitrary window size. However, we plan on adding additional complexity which allows for abstract, hierarchical, and generative constraints. Abstract constraints can be specified in a generic manner that does not require any information about the specific object below it, merely symbolic properties. For instance, we should be able to express an constraint idea such as "these bolts are have a position above the surface that is a function of both the width of the surface and the proximity of the surface to another object." Hierarchical constraints can be composed with boolean operators to create new higher-level constraints. Finally, generative constraints are constraint functions that create other partially parameterized constraints based on some variables.

Additionally, we will provide mechanisms which allow for a symbolic description of several 3D primitive shapes. The variables by which these shapes are parameterized, rather than the coordinates of their instantiated vertices, will

be symbols on which most constraints operate. Taking a cue from Aristotle, we will refer to these objects as "forms". Forms are the basic data structure of our system and are the objects with symbolic meaning that we will frame constraints around.

2 The Language

In our system, every high level constraint can be described in terms of: forms, basic constraints, compound constraints, and generative constraints. These concepts go from least general to most general, and obviously it is preferred to use simpler concepts if it is possible to explain the constraint fully.

2.1 Form

Abstract objects that have properties. These would be implemented as a tagged property list. The tagging allows us to dispatch on form type.

2.2 Basic Constraint

A function with no side effects that takes forms as inputs and returns a function that can take bindings and results in a floating point state between 1.0 and 0.0 where 1.0 is fully satisfied and 0.0 is fully not satisfied. If needed, context information for why a constraint is in a particular state may be provided. When instantiated, the forms ought be able to fully describe the 3D vertices of the object (positional information is relative to the origin; only the solver has any concept of absolute position).

2.3 Compound Constraint

A function with no side effects that takes other constraints as inputs and returns a function that combines those lower level constraints into a compound output. These would be implemented via simple boolean propagators. When lower levels are instantiated, they will propagate up to compound constraints.

2.4 Generative Constraint

A function with no side effects that takes constraints or forms as inputs and returns a function that is partially parameterized by those inputs. The result would be a function that represents the overall constraint tree. This yielded function would not return a state, but rather a partially specified constraint tree.

3 Constraint Propagation System

Our constraint system will rely on a mechanism similar to how the propagators in class work. In particular, when a higher level constraints is created, such as a compound or generative constraint, all dependent constraints must be informed that they need to propagate changes to that higher level constraint upon changes in underlying state. Basic constraints are responsible for informing higher level

compound constraints when underlying properties change. Convenience functions will be provided that allow changes in the constraint propagation to occur during simulation such as the ability to add or remove constraints from higher level constraints.

The simulation will occur within the context of a driver, similar to the drivers we have seen in many of the AMB assignments. This driver would be responsible for allowing the user to enter new data or bindings and would drive the propagation of that data through the constraint network. The driver would need to keep a lookup table of all basic constraints that rely on particular symbolic values, and when the user updates those symbolic values (such as wind direction), the driver updates those symbols. This update will trigger propagation to higher level constraints, which will ultimately result in a solution as is specified below in the Constraint Solving System section.

There is no notion of time within this constraint propagation system, and there are no convergence guarantees, but the code will be written such that the state of any given constraint can be inspected at any given moment.

4 Constraint Solving System

The input to the solver will be specified as a several constraint functions which are parameterized in terms of "forms" from the global scene.

Our solver will support a number of implementations, referred to as "modes," which are somewhat analogous to the layout types of 2D GUI frameworks.

The solver will use the following algorithm:

1. Generate inputs either via algorithmic means or static means (see stretch goals).

2. Iteratively apply constraints to the objects in the world. The action taken when a constraint fails is implementation/mode specific. Rely on constraint propagation to actually propagate changes through the network. Backtracking, if needed, will occur at the constraint level.

3. Yield for user input.

In "strict" mode, the solver will backtrack as all constraints must be satisfied. This is exponential in complexity, so we will provide alternative implementations that allow users to specify weights or priorities for constraints. Solver modes can be composed so that a subproblem can be solved with the strict implementation while another is solved with the weighted implementation.

The final output of this system will be visualized in OpenGL infrastructure already developed for the DRC. Care will be taken to ensure that the two systems are completely decoupled; the solver might output to a platform-independent CAD/modeling file which contains the final vertex lists for the output objects. Ideally, however, the output pipeline will be sufficiently fast that constraint application and search space pruning can be visualized in real time.

4.1 Stretch Goals for Solver

If possible, we would prefer to create our best "form" object approximation of an input vertex list instead of using static input data. However, this is very difficult and possibly entirely too difficult for the time span.

We would also like it if the solver could answer semantic queries, such as "why is box123 at this height?" with expressions such as "to satisfy that (1) it is above the ground plane ($z > 0$) and (2) below 50% of the height of cylinder123". This is clearly very difficult, and is therefore a stretch goal.

5 Implementation Plan

We plan to implement this project in the following steps:

1. Determine exactly what kind of constraint relationships we would like to support. If additional types of constraints beyond basic and compound are needed, this is when they would be specified.

2. Build a basic test suite that will demonstrate what we eventually hope to accomplish at a textual level. These tests would specify a number of forms in a broad range of domains, and show that our constraint propagation system can solve useful problems in all of the domains. At this stage the tests would result in purely textual information, no 3D output yet.

3. Build the constraint propagation infrastructure, in particular basic and compound constraints and the propagation of new data through the resulting constraint tree.

4. Build the constraint driver which allows adhoc changes to the constraint tree and data set.

5. Create the solver, which allows multiple modes of operation. This would allow us to start answering more broad semantic queries about the state of the system with some degree of certainty without requiring a full exponential search.

6. Create the visualization that results in a 3D "world" represented by the solution to the constraints we have imposed. This would be just a basic image or rendering.

Stretch goals are as follows:

1. Integration with the actual DRC codebase.

2. Interactive OpenGL environment.

3. A NLP parser that would take a human question about why a certain constraint is or is not being satisfied and translate it into the symbolic language that we have created.

Although we think it would be counterproductive to assign specific tasks to either group member as both members will be working towards a functional final product, the basic division of responsibilities is as follows:

Isaac: Creation of test suite, implementation of generic solver supporting multiple modes, implementation of at least a strict and relaxed mode.

Joey: Creation of test suite, implementation of constraint propagation infrastructure, implementation of driver.