# Experience Report: Constraint-Based Modeling of Autonomous Vehicle Trajectories

Kennon McKeever
University of Arizona
kennonmckeever@email.arizona.edu

Yegeta Zeleke
UC Santa Cruz
yzeleke@ucsc.edu

Matt Bunting
University of Arizona
mosfet@email.arizona.edu

Jonathan Sprinkle
University of Arizona
sprinkjm@email.arizona.edu

## Abstract

Autonomous vehicles and other robotics systems are frequently implemented using a general-purpose programming language such as C++, and prototyped using domain-specific tools such as MATLAB/Simulink, and LabVIEW. Such an approach is not efficient when programming primitive motions of autonomous vehicles when considering important safety constraints, and when promoting the broad access to robotic systems through involvement of students and aspiring students who do not know conventional low-level programming languages. Aside from general-purpose programming languages, there are languages that are specifically designed to model autonomous vehicles, such as SHIFT [3], but these languages are typically for simulation purposes only. This experience report discusses the creation of a domain-specific language that allows for faster programming of autonomous vehicles while ensuring valid constraints will be met. This language generates code for multiple controllers that will operate alternatively to allow for fast and effective programming of vehicle trajectories using primitive motions. In addition to improving coding efficiency and reducing the number of programming errors, the language adds a level of abstraction so that autonomous vehicle behaviors may be generated by people with little knowledge of low-level details of the car's operation. Furthermore, this language ensures safe operation of the vehicle by enforcing a set of user-definable constraints on the output path. A main set of constraints that are applied to every generated path have been specifically chosen to enforce safe switching between controllers and prevent the planning of unsafe actions. A novel application of the language is its ability to permit users to add specific constraints for a particular path; these constraints are checked for validity after the main constraint check is performed.

***General Terms*** Domain-Specific Language

***Keywords*** Autonomous Systems, Metamodeling, Cyber-Physical Systems

## 1. Introduction

Self-driving cars, also known as autonomous vehicles (AVs), are vehicles capable of operating without human control. Autonomous vehicles have multiple sensors that assess the environment and provide feedback in order to support safe vehicular function. The development of advanced sensors, communication media, outstanding planning algorithms and control technology has empowered the advancement of AV technology. Although this achievement provides a great opportunity to enhance autonomous vehicle safety, ensuring safe trajectories while considering vehicle dynamics in dynamic environments is still an open research problem. The degree of freedom in velocity and position provide infinite possible outcomes of trajectories for a given time horizon. While it is not difficult to implement flawless controllers that are safe under certain assumptions and restrictions, it is much harder to develop an extensible controller that continues operating safely when outside of the conditions it was designed for. One solution to this issue in terms of autonomous vehicles is to add constraints on the allowable trajectories to ensure that the controllers operate within the safe range that they were designed for. This can be achieved in a clean and effective manner with the help of a domain-specific language.

A domain-specific language (DSL) is a programming language that offers appropriate notations and abstractions for a particular domain application. Designing a DSL requires a careful consideration of the underlying application domain. A well designed DSL yields an increase in reliability, optimizability, portability, testability and consistency [13]. Furthermore, a DSL allows domain experts to remain as experts in their domain, without the need to become experts in other domains required in a compositional system.

In this paper, we explain how we defined a domain specific modeling language using a Generic Modeling Environment (GME), MetaGME. We begin by summarizing our approach and presenting relevant background information. Next, we describe our approach in-depth with examples of our solution. Then, we describe our approach to designing controllers used within our project and how we can guarantee safety with these controllers. Finally, we discuss the education applicattions and extensibility of our research.

## 2. Background

The popularity and usage of domain-specific languages has been growing over the years; in the past few years, they have attracted attention from researchers due to their various applications. DSLs are different from general-purpose programming languages in the sense that they are specialized for a specific domain or problem, such as HTML for the creation of web pages [14], SQL for developing queries on relational database [8], data storage and exchange languages such as XML and many others. DSL programs are relatively synced, self-documenting, and have various use cases [6]. A well formed DSL is capable of optimizing a system at domain level. Despite their many benefits, the limited availability of DSLs and the difficulty of obtaining valid scope for a DSL are major challenges when considering DSLs [5, 14].

Programming autonomous vehicles is generally done with low-level general-purpose languages and middleware. Although there are standards to the interface of the autonomous vehicle, such as the Joint Architecture for Unmanned Systems (JAUS), and there are methods to send JAUS command using Robot Operating System (ROS) [11], there is no standard language used to program these vehicles. Safety is a major concern in autonomous vehicles, and simple software mistakes may be catastrophic when implementing new control software. A single system may be developed and be provably safe for a certain behavioral model, however later changes to the model may require further tests to be performed. High scalability in the development of safe autonomous vehicles is not achievable when using such low-level languages. Instead, small primitive behaviors may be designed to be provably safe. Domain-specific modeling languages may further be used to combine such primitive behaviors to form a more complex model and quickly generate code for any high level changes. Some behaviors may not be combinable depending on the state of the vehicle, so primitive motions may have constraints that must be met before being performed. The domain-specific modeling language may also perform constraint checking to ensure that constraints are not violated for any primitive motion.

## 3. Method

### 3.1 Primitive Motions

In order to effectively allow for the programming of an autonomous vehicle at a high level with a modeling language, we need a way to represent a set of primitive motions of the car. For this language, we divide the car's motions into straight and turn types. These were defined as primitive motions within our study because they are very basic operations of the car that are easily understandable by many people.

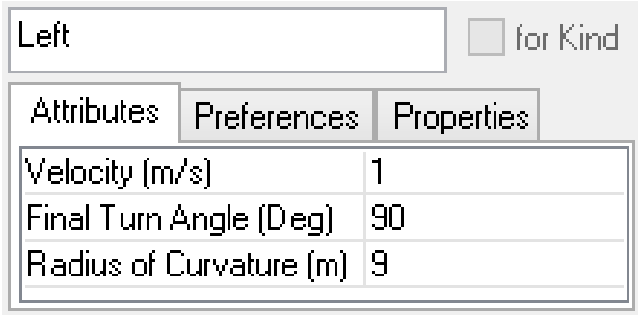


Figure 1: Attributes of a path piece with default values

In order for more complex trajectories to be designed, these primitive motion types were given attributes to modify their behavior. All path pieces have a velocity, which is the desired final velocity of the path piece. Turns have a radius of curvature and final turn angle (as shown in Figure 1) to describe the arc taken by the vehicle. Straight path pieces, on the other hand, contain a distance attribute. These attributes come with default values to allow for rapid generation of very simple paths; however, it is more useful to combine multiple primitive motions with set attributes to form more complex paths.

### 3.2 GME

In order to implement our own domain-specific modeling language, we used a generic modeling environment called MetaGME [7]. This is an environment that essentially allows the user to create a modeling environment using predefined components. Then, the created modeling environment can be registered and used within GME. This new environment now forces the user to follow a specific design pattern and syntax. Additionally, the modeling environment can call interpreters created in C++ that can be used to perform virtually any desired task.

Our implementation of the DSML in MetaGME is shown in Figure 2. From the metamodel, it is evident that paths may contain the primitive motions discussed above in section 3.1. Additionally, path models can contain other path models. This is important because it allows for paths to become new building blocks that can be used in future paths. (See Section 3.3)

### 3.3 Functional Programming

In order to ensure that larger projects are as manageable as small projects, paths are able to contain other paths, as shown in Figure 2. This is analogous to using functions in a functional programming language such as C. Although some

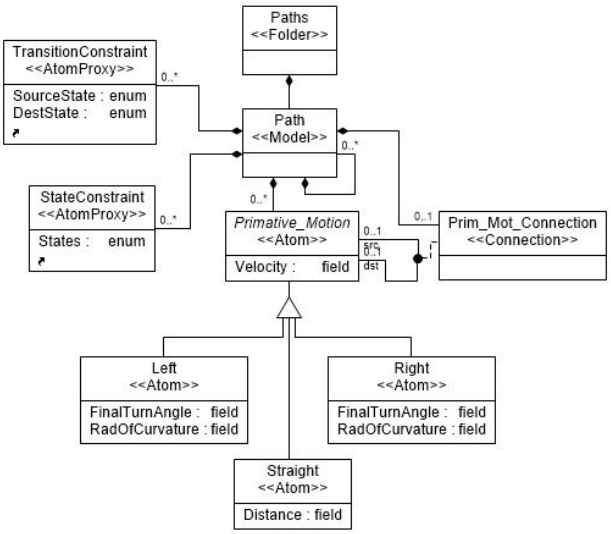


Figure 2: Metamodel of our DSML

paths need to be defined purely out of primitive motions, path models can be built using other existing path models. Thus, the user can continue to work efficiently as a project grows larger. As seen in figure 3, both primitive motions and a defined path are used to define a vehicle trajectory.

### 3.4 Metaprogramming

In programming, there are often repetitive tasks that come up and must be done repeatedly throughout the span of the project. In order to speed up development times, it is common to resort to metaprogramming. Metaprogramming allows for programs to be generated from within another program. For instance, our constraint language is able to take short high-level program statements as inputs and generate MATLAB software to check the programmed path's constraints. That way, the programmer is saved from having to write the setup code, and the overall program is produced more quickly and with a much less chance for errors.

For this particular work, we used metaprogramming in the form of code generation. When the interpreter for our DSML is run, it uses a template and sub-templates to generate a MATLAB script that checks all of the constraints defined for the described path. This approach was chosen because template-based code generation is verifiably syntactically correct [1] and it is easy to update the templates to change or add functionality of the code generator.

### 3.5 Intermediate Path Format

In addition to the domain-specific modeling language created in MetaGME, we chose to generate a small file to represent the path in what we called the Intermediate Path Format. Although we could have generated the controller code directly from our MetaGME interpreter, we decided that direct creation of the controller code would be less flexible. By using an intermediate format, the controllers and modeling environment can be operated independently. Just as how intermediate languages have been created to reduce the amount of effort needed in developing new programming languages and supporting new architectures [9], our prototype Intermediate Path Format will allow easy re-use of the controllers and modeling environment we've created in future projects. (See Section 5.2)

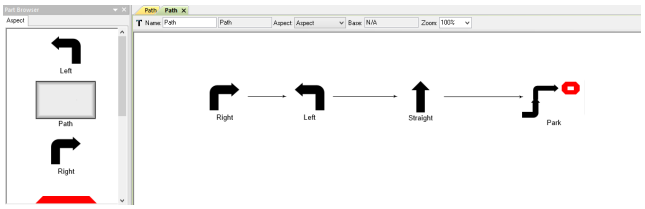


Figure 3: An example path defined in our DSML

### 3.6 Checking Path Safety

In addition to the path paradigm in GME, we have created a constraint language that allows the user to add restrictions to each of the defined paths. The constraints defined in this language can be applied to a single primitive motion type, multiple primitive motion types, or transitions from one primitive motion type to another, as in Figure 4. As an example, a constraint could be used to limit the maximum velocity of the AV in any given primitive motion type, as shown in Figure 5.

After the constraints are defined, an interpreter in GME is called that checks the states of each primitive motion and makes sure that none of the defined rules are violated. If the interpreter encounters a rule violation, it prints the error and refuses to generate code to run on the car, as shown in Figure 6 where the maximum velocity constraint from above was violated. By treating a failed constraint condition as a compile error, the user can be certain that any code generated is verified as safe with respect to both our predefined constraints and user-defined constraints.

These constraints are useful because planning paths using primitive motions can lead to unsafe behavior. For example, the user may use primitive motions to define a path where the vehicle is commanded to accelerate to a high velocity and take a sharp turn. With the constraint language, it is possible to prevent the user from describing unsafe actions for the car by simply preventing primitive motions from being combined in an unsafe sequence. Moreover, the constraint language is also useful because it can apply restrictions on primitive motions that will aid in ensuring controller safety, such as preventing states that would be unsafe in switching control.

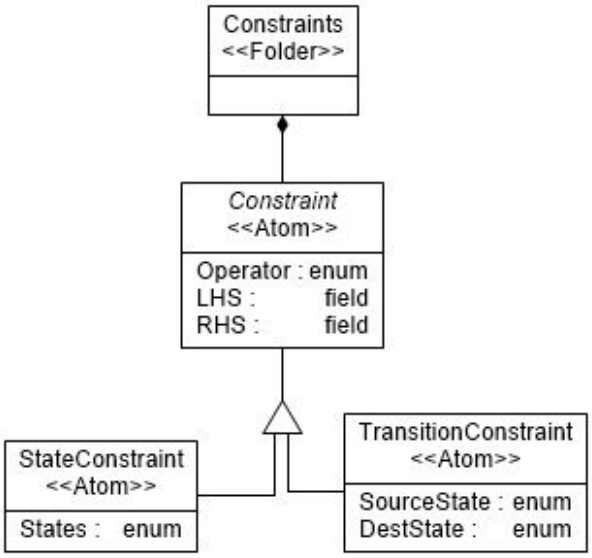


Figure 4: Metamodel of the Constraint Language

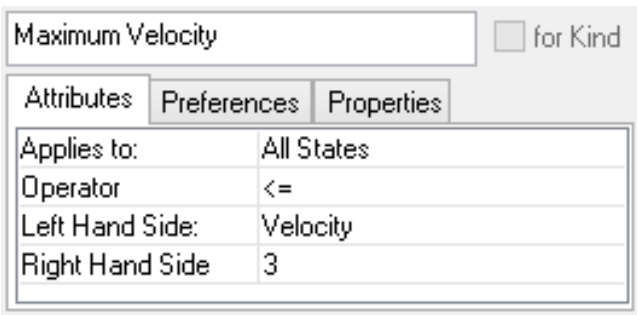


Figure 5: An example of a path constraint

```
>> Path_Constr
failed state constraint Maximum Velocity at State #2
```

Figure 6: An example of a constraint violation preventing code generation

# 4. Switching Control and Switching Control Safety

## 4.1 Switching Control

Switching control is the use of multiple controllers or algorithms operating alternatively to control a switched system [15]. These different controllers are typically specialized to perform well for certain conditions within a system. In order to operate effectively, these different controllers are managed by a governing rule that handles which controller is operational at any given time based on the system's circumstances.

We decided that switching control was an effective way to implement controllers for our modeling language. We designed a controller for each primitive motion described in section 3.1. The transitions between these controllers then corresponded with the natural transitions defined in the user's sequence model of the vehicle trajectory, yielding a simple way to define the rule used to govern the individual controllers.

In addition to these natural transition points, it is important to note that the implementation of certain controllers may require additional transitions. For example, one might choose to add more controllers that handle changing speeds more effectively than the basic controllers. In this scenario, one would have to identify what conditions justify switching to these new controllers and when these conditions occur. In our implementation however, there was no need for these intermediary controllers because we prevented such conditions from occurring due to the constraint language.

## 4.2 Switching Control Safety

Our system is guaranteed to be safe in the sense that it is impossible for the system to transition into an invalid state. An example of an invalid state in this sense would be taking a sharp turn immediately following a piece of the path with a high velocity. As described in [2], there is an algorithm to find safe switching surfaces within our state space. This can be done in future work but our current implementation involves very basic controllers, so we guaranteed that the controllers remain in safe sates by imposing restrictions on generated paths with the constraint language described in section 3.6.

# 5. Applications

## 5.1 Use In Education

As mentioned previously, our modeling environment is extremely powerful in the sense that paths can be planned out and generated in a matter of minutes. Like most DSLs, the high-level nature of the language means that it can be programmed more easily by a larger group of people [10], including people that are not experts in control systems. Thus, high school students will be able to use our path-planning language to design vehicle trajectories with minimal knowledge of how the car works. Even without understanding the mathematics used to calculate the primitive motion attributes, they will still be able to design very simple paths using the primitive motions' default attribute values discussed in Section 3.1.

## 5.2 Project Extensibility

Due to the simple nature of the intermediate path format described previously, our environment can easily be used in the development and testing of other programs. For example, one would be able to test a path-finding algorithm on an autonomous car as long as the algorithm can output the


(a) Autonomously following trajectory.

(b) Live demo using the modeling language.

Figure 7: (a) The autonomous vehicle following the trajectory synthesized by the modeling language. (b) One of the authors demonstrating the language use for a reporter at the live demonstration.

desired primitive motions. This would prevent the user from having to write their own controllers or learn how the vehicle operates. Additionally, the global constraints will make sure that the algorithm comes up with a feasible path.

The use of the intermediate path format also allows for faster development of new controllers for our vehicle. Although we have developed basic controllers to work with our DSML, one might choose to update these controllers to better suit their needs, such as improving primitive motion accuracy or smoothing transitions between primitive motions. As long as these primitive motion controllers are designed to accept their inputs from the intermediate path format, our DSML and global safety constraints can be used to quickly design safe new paths used to test these controllers. Most importantly, the constraint language can be used to impose further limitations on path design that will guarantee safe conditions for each individual controller and valid transitional states in switching control.

Finally, the intermediate path format can be used to allow our path-planning modeling environment to be used by other vehicles. Since the controllers we have created are not necessarily compatible with all autonomous ground vehicles, new controllers that operate on the intermediate path format that are made for a different vehicle can be used in place of our basic controllers. Then, the general primitive motion restrictions defined in our constraint language could be updated to reflect the attributes of this new AV. The end result is that the DSML that we've created will essentially be extended to work with this new vehicle as well.

## 6. Demonstration and Results

On August 11, 2015, the modeling language was used to demonstrate the CAT Vehicle in front of an audience of graduate students, faculty, visitors to the University of Arizona, and various broadcast media. In the live demonstration, two different trajectories were demonstrated:

1. a trajectory that performed a figure-8 style maneuver around several predefined obstacles; and
2. a trajectory of straight and turning primitives, which was designed to end in exactly the same place where it began.

In Figure 7 images of the live demonstration are featured. In these images the vehicle is under autonomous control of the synthesized software from the modeling langauge, and the output code is verified to obey each of the runtime physics-based constraints using the approach defined in previous sections.

The results were robust enough to provide "rides" in the vehicle to visitors, and some of the rides were featured in stories covered by the various broadcast media outlets [4, 12].

## 7. Conclusion

In this article, we have introduced a domain-specific modeling language for programming autonomous vehicles. This is a much more effective than current methods for programming AVs because the DSL is much more expressive than a general-purpose language. Additionally, larger projects are easy to manage because the DSML allows for metaprogramming.

The constraint language included is also a crucial part of the DSML because it allows the programmer to guarantee the safety of a trajectory using the predefined set of constraints. It also allows for the creation of user-defined constraints that can guarantee custom controller safety, guarantee safe operation of a different AV, and offer adaptability to the needs of the testing environment.

## 8. Future Work

Due to rain in the live demonstration, the approach to utilize dead reckoning localization resulted in several behaviors that strayed from their intended trajectories. Although the vehicle never violated any of its safety constraints, these constraints did not include some obstacles that are not accounted for *a priori*, so integration of various additional sensors is an important safety feature to add to the system setup.

Further, the trajectory synthesis depends on knowledge of local sensor values and integration of those values to achieve distance estimates: future efforts may utilize global sensors (such as GPS) to transform local estimate goals into projections into the global sphere, in which case global sensors can be utilized to ensure that heading and distance traveled can be accurately computed.

### Acknowledgments

This work was supported in part by the National Science Foundation and the Air Force Office of Scientific Research, under awards IIS-1262960 and CNS-1253334. Thanks to other members of the CAT Vehicle 2015 REU Site at the University of Arizona for their help in testing the vehicle, and providing safety supervision for the test area on the date of the demonstration.

## References

[1] J. Arnoldus, M. Van den Brand, A. Serebrenik, and J. J. Brunekreef. *Code generation with templates*, volume 1. Springer Science & Business Media, 2012.

[2] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers for linear systems. *Proceedings of the IEEE*, 88(7):1011–1025, 2000.

[3] A. Deshpande, A. Göllü, and P. Varaiya. *SHIFT: A formalism and a programming language for dynamic networks of hybrid automata*. Springer, 1997.

[4] S. Donaldson. Look ma – no hands!, Aug. 2015. URL `http://ece.arizona.edu/look-ma-no-hands`.

[5] C. W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[6] D. A. Ladd and J. C. Ramming. *Two application languagesin software production. In USENIX Very High Level Languages Symposium Proceedings*. 1994.

[7] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, page 1, 2001.

[8] D. Leijen and E. Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[9] S. Macrakis. From uncol to andf: Progress in standard intermediate languages. *Open Software Foundation, macrakis@ osf. org*, pages 1–18, 1993.

[10] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[11] P. Morley, A. Warren, E. Rabb, S. Whitsitt, M. Bunting, and J. Sprinkle. Generating a ros/jaus bridge for an autonomous ground vehicle. In *Proceedings of the 2013 ACM workshop on Domain-specific modeling*, pages 13–18. ACM, 2013.

[12] M. Popat. Hands off the wheel, please, Aug. 2015. URL `http://tucson.com/news/local/take-a-ride-in-ua-s-driverless-car/youtube_00f86f2c-408f-11e5-b94f-3bb8d0df702b.html`.

[13] A. Van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, 10(1):1–17, 2002.

[14] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35 (6):26–36, 2000.

[15] E. Zuazua. Switching controls. *Journal of the European Mathematical Society*, 2008.