A Safe Autonomous Vehicle Trajectory Domain Specific Modeling Language For Non-Expert Development

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

Yegeta Zeleke
UC Santa Cruz
yzeleke@ucsc.edu

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

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

Abstract

Managing complexity while ensuring safely designed behaviors is important for cyber-physical systems as they are continually introduced to consumers, such as autonomous vehicles. Safety considerations are important as programming interfaces become open to experts and non experts of varying degrees. Autonomous vehicles are an example system where many domain experts must collaborate together to ensure safe operation. Through the use of higher level abstraction, domain experts may provide verification tools to check dynamic behavioral constraints. Similarly, higher level modeling tools may generate lower level artifacts for a working system. With modeling and verification tools, smaller teams and potentially non-experts may program custom behaviors while ensuring a correctly behaved system.

A high level domain specific modeling language was created with a focus on non-domain experts. The domain consists of driving a vehicle through a set of known waypoints by connecting together multiple primitive motions in a sequence. Though constrained to simple motions, it is still possible to create a sequence to drive the vehicle unsafely. Model verification was implemented to check that the expected start and stop waypoints were correctly reached without driving the vehicle into unsafe regions. The language was then provided to a set of 4th year elementary school students to create unique paths. The models created by the students were then used to generate controller artifacts to operate a real autonomous vehicle on a soccer field.

Keywords Domain Specific Modeling, Cyber Physical Systems, Model Verification, Metamodeling

1. Introduction

The design of behaviors for cyber-physical systems is typically a complex process involving a collaboration of efforts between experts in various domains. Development becomes unattainable for many people such products, even if ideas of new behaviors are conceptually easy to understand at a high level. Autonomous vehicles are an example of such a system that is becoming increasing mainstream, where safety is one of the largest concerns.

Both experts and non-experts may want to design a custom behavior for safety critical systems. The use of a domain specific modeling language (DSML) is a way to constrain behavior designers to a specific modeling structure. A structurally constrained solution however does not necessarily translate to a dynamically safe solution, thus users of a DSML still need to carefully evaluate the design. Non-experts may not be fully knowledgable in the domain and all safety factors, while domain experts may easily make mistakes during programming or may forget to consider all scenarios. Fortunately the use of a DSML means that model tools may be implemented to both translate models to working solutions on a target platform, and to perform model analysis to ensure that only working solutions may be generated.

Generated code from models ensures that behavioral designers do not need to focus on low level syntax and semantics, reducing overall development and modeling language training time. Smaller development teams, potentially individuals, may also be capable of quickly trying out new behaviors while feeling confident that only safe designs will be produced. The implementation of verification tools is a way of including pair programming techniques, where the domain expert has previously implemented a way to check for all possible design faults. Such a domain expert may be able to create a modeling language in such a way that only safe artifacts will be generated under the use of non-domain experts. With an automated way of providing verification failure feedback, the original verification tool creator has a way to communicate issues in the design to the modeler.

This project places a focus on providing a modeling tool tailored for 4th year elementary school students at a Canyon View elementary school enrolled in a Lego based robotics course. Since the students had prior experience in using a visual modeling language for the Lego NXT (Kim and Jeon 2007), the students should be capable of using other visual based languages with a small training duration. Verification tools were also implemented so that students could check for their path correctness. The verification tool was designed to check for all aspects concerning safety of the vehicle in a particular design domain using the language.

2. Background

The use of metamodeling for designing DSMLs enables easy rapid prototyping by defining the structure of the language by defin-

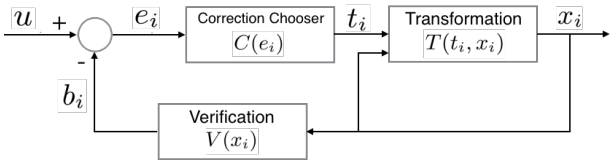

Figure 1. Model verification with automated correction loop based on dynamic constraints.

ing a domain specific syntax. The Generic Modeling Environment (GME) is a tool that implements metamodeling for easy development of visual languages (Ledeczi et al. 2001). Evolution of a DSML is easily possible through GME's modeling language generation support. Various add-on components may be included to perform various artifact generation and model checking. A commonly used add-on in GME is an interpreter which may transfer models to usable various artifacts. Code generation is an example of a generation which features various benefits such as confidence in generating syntactically correct code (Kelly and Tolvanen 2008).

Other tools that may be included are model analysis tools. The use of verification results in confidence of the model (Thacker et al. 2004). Before generating artifacts from models, it may be possible to ensure that models are first verified. Generated artifacts may then be proven to have correct or safe behavior under known model verification (Whitsitt and Sprinkle 2013).

The particular verification techniques are domain dependent. Prior DSMLs have implemented off the shelf verification tools, such as for state machine checking (Zhang and Sprinkle 2014). Knowledge of verification failures may be used to modify models to ensure that verification will pass. Some work on model verification also takes advantage of model transformation to automatically correct models (Whitsitt 2014) (Zhang and Sprinkle 2014).

During the summer of 2015, a Research Experience for Undergraduates (REU) program held a group that built a modeling language in GME for defining vehicle trajectories (McKeever et al. 2015). The work also implemented verification techniques to ensure that vehicles always be safely programmed under a particular safety definition. The work involved the design of a sequence of primitive controllers to be invoked for a vehicle while parameterizing velocities, driving distance, and turning curvatures. Though the safety definition was verifiable for the modeling language domain, further safety definition refinements require rework to the verification process for modified domains. This prior work serves as the base effort for the described project, with modifications made based on modified design domain.

3. Model Correction Framework

The use of a modeling language places constraints on the structure of models, however typically does not perform dynamic analysis during design time. Modeling languages with interpreters and verification tools have the capability to produce dynamically correct output artifacts. A correct working artifact is one that meets a set of dynamic constraints, defined either implicitly by the modeling language or explicitly by the modeler. Figure 1 illustrates a discrete feedback system for automatically correcting model dynamic behavior using model verification and model transformation. Provided a set of constraints to be met u, an initial model x_0 may be provided. The model at any given time x_i undergoes a transformation to fit a model verification tool $V(x_i)$ to determine the dynamic behavior of the model b_i. Comparison between the desired behavioral constraints u and the true behavior b_i produces constraint violations denoted as model error e_i. Ideally e_i should eventually represent an empty set of constraint violations. Analysis performed on e_i may be performed to determine the proper technique required to correct the model removing the constraint violation, denoted at the design technique chooser $C(e_i)$. The resulting technique t_i and prior model x_n may then be provided to a model transformation tool $T(t_i, x_i)$ to generate an updated model x_{i+1}.

A framework using this model may be deployed at various levels of automation. If a modeling language can be tightly coupled with verification and transformation tools, a modeling language potentially has the capability to always generate working models. Each system and language may vary greatly, and therefore each automated model correction feedback loop may only produce models under certain initial states and dynamic constraints. The design of the model transformation and correction chooser are the key elements of design for a stable correct model generation.

4. Language Design

Prior work done in GME involving a sequence model design was used as the main inspiration for this project. Since the work already provided controls based on a sequence model, a similar language should be designed so that the design space matches the application space. Before modifying the language, coordination with the planned deployment group to define a short curriculum based on the student's capabilities. Due to the focus on education for the students, the modeling language is a simplified version of the prior path planning work, and feedback was provided directly to the students to correct their own models. This means that referring to figure 1, students assumed the role of both the correction chooser and model transformation block.

4.1 Example Behaviors

Before designing a modeling language, specific example paths were designed to carve out how the students should design their own paths. Focusing on the application first ensures that little changes will be required in the modeling environment, and thus the metamodel. The prior work included a set of Matlab based controllers that imported a configuration file containing the path sequence. The path sequence had various parameters including the ability to set the vehicle velocity, distance to travel, turning curvature, and final turn angle for commands of driving straight, right and left. Such parameters were deemed to be potentially too difficult for the student's math capability, therefore a predetermined set was created for a particular set of waypoints.

Students were tasked with measuring the final demonstration area for the known drivable space, which was located in a soccer field at the elementary school. The desired behavior would then only be capable of driving to a point in front to the car, to the front right, or to the front left. Initially the plan was to set a grid of 3x3 waypoints, however the possible solution set did not allow for many creative paths and was therefore increased to 4x4. Each waypoint was then configured in a square grid with a $10m$ distance between points. With the known set of waypoints and dimensions, it was clear to set the straight controllers to only drive for a distance of $10m$, and to have the turn controllers drive the vehicle at a radius of 10m and only turn at angles of $90°$. The velocity was set as a constant for all controllers to be $1m/s$. With the known possible set of commands in a sequence model, the controllers could be modified to only ever use the set of predetermined parameters, thus ensuring that paths were safely constrained.

Figure 2 shows the map created by the students along with the waypoint set, waypoint labeling, and an example path. For more creativity, support was implemented for arbitrary path start and stop points. Each path start and stop point could be defined by the column, row, and cardinal direction. During the demonstration day, it was planned to manually drive the car to the start location with the proper orientation, then start the path.

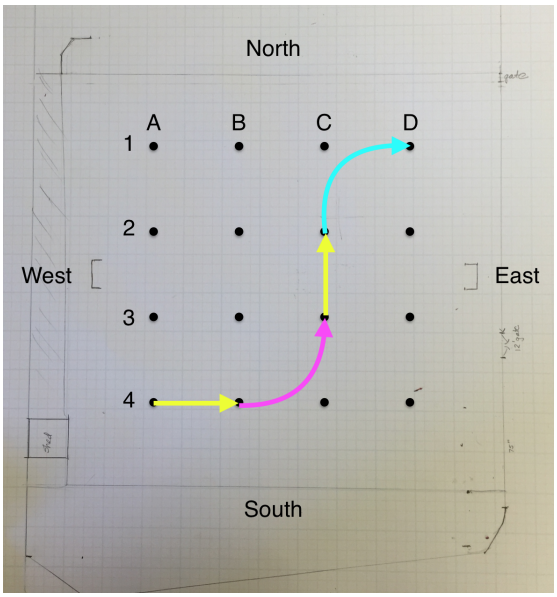

Figure 2. The grid design space for the students on a hand drawn map. Students measure out the driving space, and a grid was creating with column and row labeling, orientation, and an example path with straight, left, and right commands.

4.2 Modeling Environment

In the prior sequence model designer, GME was the main tool which was operating system specific and requires additional plugins to be installed for model interpretation. Since the students would all be working on computers supplied by the school on an unsupported operating system, the Web Generic Modeling Environment (WebGME) tool was chosen (Maróti et al. 2014). WebGME offers the same metamodeling concepts as GME, but also includes other features such as account management and an operating system agnostic interface. During the time that students would program their models, a WebGME server was set up offsite and accessible through the internet. Since students were working in groups, accounts were created beforehand for each group so that groups could log in to their own account and begin modeling.

4.3 Metamodel

With the known possible generated solutions, an adapted version of the prior work's metamodel was implemented. Figure 3 shows the design, where a path is comprised of a set of primitive motions that are connected by an ordering denoted as Prim_Mot_Connection. The only set of concrete primitive motions are straight, right, and left. Though the prior work used the attributes in the concrete motions, it is noted that the controllers were customized to ensure that any change in the attributes would not be reflected in the vehicles behavior.

The converted metamodel is nearly identical except for an additional set of attributes for setting the start and stop waypoints. Enumerations were used so that the students could only select 1-4 for Y, A-D for X, and N, S, W, or E for Direction. The start and stop waypoints are used as explicit behavioral constraints for the path, to ensure that the vehicle will start and stop provided the sequence of commands.

4.4 Modeling Language

The metamodel is used in WebGME to directly generate the structural constraints of the modeling language. Figure 4 shows the interface of WebGME with an example model. The interface implements a parts browser on the left for students to drag in commands. After dragging in various command components, students were able to connect commands to each other. The lower right shows a set of attributes corresponding to the start and stop waypoints for the particular path. WebGME also implements a way to invoke various custom plugins, such as interpreters or listen to object modifications.

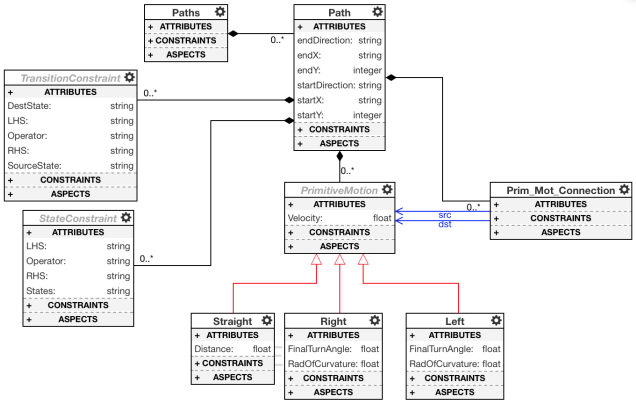

Figure 3. The metamodel for designing paths.

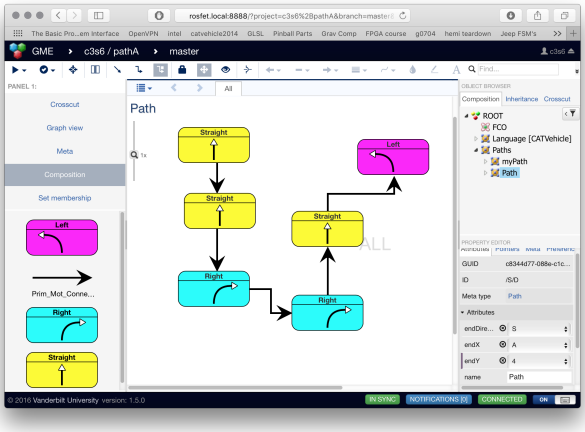

Figure 4. The modeling interface showing a path built by a student group. The left shows a parts browser for drag and drop based path creation with decorated primitive motions. The lower right shows attributes of start and end waypoints.

4.5 Decorator

Often the default modeling elements created for the modeling language from the metamodel are only text based. As a better visual aid for the students, a decorator plugin was developed to help students show the shape of motions being connected. Figure 4 shows an example decorated path model. Each command shows an arrow demonstrating the motion that the vehicle will take. The commands were also colored for easier path debugging. The sequence of motions was defined with the Prim_Mot_Connection, with an arrow designating the order.

4.6 Model Verification

The model of the paths underwent a few different structural and behavioral verification steps. Verification must pass before generating controller artifacts to ensure that the model both fits the controller's domain, and to ensure that the path does not violate behavioral constraints.

4.6.1 Sequence Structural Integrity

During the creation process, vehicle commands may be created before the ordering is set with a Prim_Mot_Connection. Similarly if a modeling mistake was made, multiple connections were allowed during modeling. Before generating artifacts however, multiple incoming or multiple outgoing connections were not allowed since the Matlab controllers do not support path branching. Also, it is structurally possible to create multiple paths upon deleting intermediate paths or connections; a necessary part of path editing. This creates ambiguities on which path should be used for generating artifacts, and is not allowed on interpretation. Lastly, cyclic paths are possible to model, but are not supported.

To check for all of these conditions, pseudo code is provided in algorithm 1. If this test fails, a message is given to the user upon invoking the interpreter stating the particular detected error, and no artifacts are generated.

Algorithm 1 Check Path Integrity

```
1:  procedure GOODPATHINTEGRITY
2:      fail = false
3:      nStart ← 0
4:      nEnd ← 0
5:      nBranch ← 0
6:      for motion ∈ PrimitiveMotions do
7:          i = i + 1
8:          if getNumSrcConnections(motion) = 0 then
9:              nEnd ← nEnd + 1
10:         else if getNumSrcConnections(motion) ≠ 1 then
11:             nBranch ← nBranch + 1
12:         if getNumDstConnections(motion) = 0 then
13:             nStart ← nStart + 1
14:         else if getNumDstConnections(motion) ≠ 1 then
15:             nBranch ← nBranch + 1
16:     if nStart ≠ 1 ∨ nEnd ≠ 1 ∨ nBranch ≠ 0 then
17:         fail = true
        return ¬fail
```

4.6.2 Dynamic Constraint Verification

A well defined path structure does not mean that the vehicle will behave as expected. For example, an unsafe path would be to command the vehicle to drive straight more than 3 times, driving beyond the bounds of the defined grid. Because arbitrary start and end points were allowed, a path integration algorithm was implemented to ensure that the vehicle program would terminate at the defined end waypoint and direction and also would also not drive out of the grid bounds. Since no student should create a model that drives the vehicle out of bounds, the boundary constraint was defined as an implicit constraint for the language. Algorithm 2 demonstrates the pseudocode for checking the path's behaviors based on a given start and and point.

4.7 Feedback

The previously described pseudocode was implemented in WebGME using the JavaScript based plugin framework. Not described in the algorithms is the error handling and feedback to the user. Modeling error feedback is an important part during development

Algorithm 2 Check Path Behavior

```
1:  procedure GOODPATHBEHAVIOR
2:      motion = getStartMotion()
3:      waypoint = getStartWaypoint()
4:      repeat
5:          waypoint ← integrate(motion, waypoint)
6:          if ¬inBounds(waypoint) then return false
7:          motion ← getNextMotion(motion)
8:      until getNumSrcConnections(motion) = 0
9:      waypoint ← integrate(motion, waypoint)
10:     if ¬inBounds(waypoint) then return false
11:     if waypoint ≠ getEndWaypoint() then return false
        return true
```

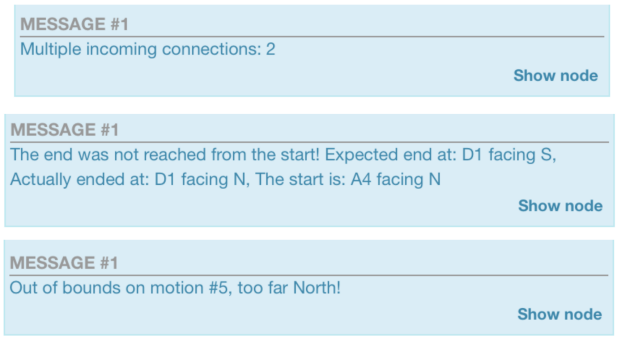

Figure 5. Verification failure feedback in WebGME showing three different errors. Top: Branching error. Middle: Incorrect end waypoint orientation. Bottom: Driving out of bounds.

specifically for describing the particular errors in the model. The lack of such a description would require users to carefully reevaluate the entire design rather than the specific problem area. For Elementary school students, the type of feedback is important to help their spatial understanding of what the path errors were.

Figure 5 shows messages from three verification failures. The first error demonstrates poor path integrity, in this case one motion with two incoming arrows. The second error had an incorrectly defined ending waypoint orientation. Here the true final location and orientation is shown for the students to compare to what they expected. The third error message shows that at the 5th motion in the sequence, the vehicle was commands to drive out of the drivable area, as well as which side of the map the vehicle would try to drive. The students can take this message and perform their own path integrating with their model to determine the modeling error.

4.8 Model Interpreter

Once a model is known to be both structurally and behaviorally correct, the model can be interpreted into working and safe artifacts to operate the vehicle. The interpreter was ported from the prior GME based work to WebGME with little modifications. The output artifacts of the interpreter is a parameters file importable by the Matlab controllers, containing the sequence of controllers to be invoked along with controller parameters. Invoking the interpreter occurs in the same step as the model verification to ensure that only verified models are interpreted.

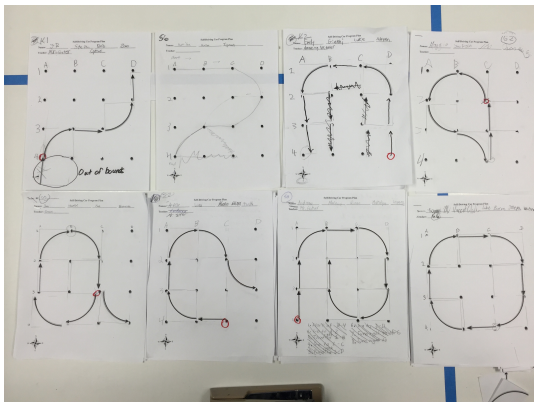

Figure 6. Tiled paths by 8 selected groups during the path planning session.

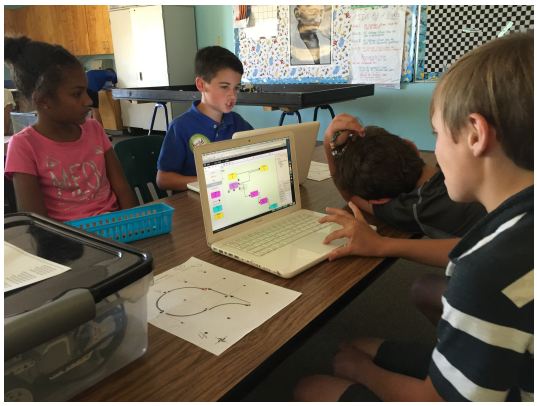

Figure 7. Student groups transferring the tiled paths to a DSML model.

Figure 8. Running one of the student's paths on the CAT Vehicle.

5. Deployment

5.1 Preliminary Path Planning

Prior to the day dedicated to programming, students were tasked with coming up with paths so that they did not need to focus on the creativity aspect while also programming. Once the student had a path, the transition to the modeling language should be more clear. Students were provided with a grid with cardinal directions and waypoint labeling on a piece of paper. Students were also provided a set of paper tiles with the possible motions of right, left, and straight for the car. In groups of 4, students worked on creating a path by placing tiles on the grid. There were a total of 18 groups. Once a path was created, students wrote down the start and end waypoint and direction. Figure 6 shows 8 selected group's paths for the demonstration day.

5.2 Modeling

A day after the path planning, each group was given a 30 minute session for learning the modeling language and transferring the paper path into a model. Due to the size of each group, groups were split in smaller groups of 2 so that each student would have more direct interaction with the modeling language. Once students designed a possible path, the verification tool and interpreter were invoked. If the students received a warning message, the students were tasked to double check their paper tiled path and ensure that their model matched. All students were able to eventually produce a working model. Figure 7 shows one group working on transferring their set of tiles to the modeling language.

5.3 Demonstration

A day after the modeling language session, a selected set of groups were able to demonstrate their paths. The demonstration area waypoints were marked using paint. The center of each grid square (made up of 4 waypoints) was marked using orange cones, resulting a 9x9 cone grid for a 4x4 set of waypoints. Before each path was executed, the car was manually driven to the start point defined by the students. During the setup process, a student from each group described the path to the spectators. Figure 8 is a picture taken form the demonstration day, running one of the programs created by a student group.

6. Results

6.1 Path Creation

Initially there was concern about the possible complexity of motion planning for the students, even with a constrained domain. Though the students had worked with Lego based robots before, the Ackerman style steering mechanics result in a further non-holonomic type of planning. Some groups were not able to create a path on the creation day due to not understanding that a car is incapable of turning in place, unlike skid-steer style assemblies typically built in Lego. Most students were capable of understanding how the orientation integrates from the prior motion command and were able to make effective paths. Fortunately the students that were not able to make a path in the first day were able to both create a path and a model in the language during the modeling language session.

6.2 Modeling

The students had prior experience working with visual based programming languages, making the connection aspect intuitive in the path language. Most students were able to jump right into path creation and begin connecting motions to create a sequence. Transferring the paper paths to the modeling required a bit of spatial reasoning, and some students excelled better than others. A few groups created their paths perfectly on the first attempt, passing verification with ease. A few other groups unintentionally ended up creating some branching paths, and the verification tool was able to inform them of the issue.

The most common mistake made by students was understanding the difference between right and left when the orientation was not oriented north. The verification feedback would provide some sense of an error, however it would not directly say which motion was incorrect, due to there being non-unique solutions. Students were instructed not with what should be changed, but to double check their paths. Also, students were told to hold specific motions next to the computer screen, and rotate the paper until they could figure out which one matched the motion. The decorator proved to be very useful by providing a direct visual match to the paper

tile. Also, the delineation of color between the three motions aided in quickly determining incorrectly placed motions, such as when a group's path only had straight and left motions yet all three colored motions were present.

The feedback provided by the verification tool was also useful for when paths went out of bounds or when they did not end at where it was expected. A couple of instances occurred where students correctly inserted their path, but did not enter the correct start and and waypoints. One shortcoming of the feedback provided was that it assumed the start waypoint to be correct, which often would say that the path went out of bounds. Additional feedback would be to provide the set start location upon verification failures.

6.3 Running Paths

In safety critical systems, often custom programs are statically verified and simulated before implementation on a physical system. Much of the effort for the project went into the verification tools and testing the controllers. Therefore with a known working verification tool for this domain, there was no need to check the models by hand. Once the students were able to get verification to pass, the system was fully trusted to operate safely for the demo. Because verification was required to generate the controller artifacts, it was also known that unsafe files would never be provided. Therefore the prior efforts provided a great deal of safety certainty. The demonstration involved directly taking what the students wrote and running them on the vehicle. All paths worked exactly as expected, as designed by the students.

7. Conclusion

The DSML was shown to be successful by letting non-experts in the area of autonomous vehicle generate safely working behaviors. The language and verification tools are however limited to specific motions and a specific number of waypoints. Further work may involve expanding the language to have a more flexible set of motions with reactive behaviors. The students have experience working with simple sensors which may be emulated using more sophisticated sensors on the CAT Vehicle, emulating behaviors programmable in the Lego kits.

The modeling language was largely an enjoyable and educational experience for the students. Even though the students have become knowledgable in lego design and programming, the path programming still required students to use spatial reasoning to implement a correct path design. This work could be expanded further as an educational tool for other age groups and perhaps add more mathematical concepts. With the proper verification tools, more advanced students could implement more mathematic planning in a language. For example, instead of having waypoints students could be tasked with driving the car to a specific waypoint in a minimal amount of time using my programming turning radius and velocities.

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 Charlotte Ackerman, Robotics Instructor in the Catalina School District for helping to coordinate the sessions with the students. Thanks to Rob Henikman, Canyon View Principal for providing logistical support and planning.

References

S. Kelly and J.-P. Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

S. H. Kim and J. W. Jeon. Programming lego mindstorms nxt with visual programming. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*, pages 2468–2472. IEEE, 2007.

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.

M. Maróti, R. Kereskényi, T. Kecskés, P. Völgyesi, and A. Lédeczi. Online collaborative environment for designing complex computational systems. *Procedia Computer Science*, 29:2432–2441, 2014.

K. McKeever, Y. Zeleke, M. Bunting, and J. Sprinkle. Experience report: Constraint-based modeling of autonomous vehicle trajectories. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM 2015, pages 17–22, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3903-2. doi: 10.1145/2846696.2846706. URL `http://doi.acm.org/10.1145/2846696.2846706`.

B. H. Thacker, S. W. Doebling, F. M. Hemez, M. C. Anderson, J. E. Pepin, and E. A. Rodriguez. Concepts of model verification and validation. Technical report, Los Alamos National Lab., Los Alamos, NM (US), 2004.

S. Whitsitt. Automatic verification of dynamic constraints in lti control systems through model transformations. In *NSF Young Professionals Workshop on Exploring New Frontiers in Cyber-Physical Systems*, Washington, DC, 03/2014 2014. NSF, NSF.

S. Whitsitt and J. Sprinkle. Model based development with the skeleton design method. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 12–19, April 2013. doi: 10.1109/ECBS.2013.16.

K. Zhang and J. Sprinkle. A closed-loop model-based design approach based on automatic verification and transformation. In *Proceedings of the 14th Workshop on Domain-Specific Modeling*, DSM '14, pages 1–6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2156-3. doi: 10.1145/2688447.2688448. URL `http://doi.acm.org/10.1145/2688447.2688448`.