A Framework for Automatic Dynamic Constraint Verification in Cyber Physical System Modeling Languages

A FRAMEWORK FOR AUTOMATIC DYNAMIC CONSTRAINT VERIFICATION IN CYBER PHYSICAL SYSTEM MODELING LANGUAGES

by

Matt Bunting

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2020

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Matthew Bunting, titled "A Framework for Automatic Dynamic Constraint Verification in Cyber Physical System Modeling Languages" and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Jonathan Sprinkle Date: December 3, 2019

Roman Lysecky Date: December 3, 2019

Hal Tharp Date: December 3, 2019

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Jonathan Sprinkle Date: 13 JAN 2020
Dissertation Committee Chair
Electrical and Computer Engineering

ACKNOWLEDGEMENTS

I would like to acknowledge the National Science Foundation for their support under award CNS-1253334.

I am eternally grateful for my advisor Professor Jonathan Sprinkle for his mentorship during my graduate student career. He has provided incredible and unique research and academic opportunities that have directly correlated to my success. This work would not be possible without his guidance.

I would also like to thank my parents, friends, and family for their unhindered support. I would not have accomplished what I have without them.

Finally, I would also like to thank my girlfriend Ashley Ramsey. She has provided an incredible amount of close support throughout the majority of my graduate career.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF FIGURES – *Continued*

LIST OF TABLES

Listings

ABSTRACT

Design of Cyber-Physical Systems (CPSs) involves overlapping the domains of control theory, network communication, and computational algorithms. Involving multiple domains within the same design greatly increases the system complexity. Furthermore, the physical nature of CPSs generally involves important safety constraints where constraint violations can be catastrophic. The design of CPSs benefits from focusing on the construction of abstracted, high-level models in a Domain-Specific Modeling Language (DSML). A Domain-Specific Modeling Environment (DSME) may aid in the design of such complex systems by enforcing structural design constraints during the construction of models. Models built using a DSME may also use compilers or interpreters to produce real working, low-level artifacts that represent the high-level design. Though each model in a DSME may abide by a formal specification, the behavior of a design may violate dynamic constraints if deployed. Engineers are tasked to ensure that models behave safely by implementing their expert knowledge after using appropriate verification tools. Constraint violations may be eliminated by a modification of the model based on verification feedback, known as Dynamic Constraint Feedback (DCF). Mending such constraint violations is a task generally performed by the model designer. Such a process could potentially be automated through the capture of well-known design practices. The challenging task when automating model correction then becomes in the design of a DSML. A designer of a DSML may have a clear understanding of how to design the syntax and semantics for their domain, but there are no formal methods for implementing verification tools for automatic model correction. Such a framework could greatly aid in the selection of available verification tools, implement well-established design methods, and model dynamic constraints. Presented is the Dynamic Constraint Feedback Metamodeling Language (DCFML), a new metamodel to imple-

ment DCF upfront in DSML design. This particular solution provides a concrete solution to the abstraction of the various components of DCF, and then appends them to the DSML design process provided by a DSME.

CHAPTER 1

Introduction

1.1 Overview

A Cyber-Physical System (CPS) is a system that interfaces computational techniques with physical elements [59] [99]. A CPS is by nature a multidisciplinary field involving multiple design domains. Formally, a CPS encompasses the domains of network communication, computational algorithms, and control systems [11]. Each domain may provide unique techniques to solve design problems relative to other domains. A particular domain may also benefit from the integration of domain concepts from another. For example, the design of a control system to operate a physical process may benefit from the use of computational perception algorithms instead of using an analog solution. Design of a CPS is therefore typically done as an integration of multiple networked components rather than focusing on the design of a single element [68].

Various design domains may be under the umbrella of CPS classification, as long as they incorporate two or more of the fundamental domains comprising of a CPS. Such design domains may include the industries of power distribution grids [26] [117], self-driving vehicles [73] [51] [78], medical devices and monitoring [121] [83], automatic pilots in aviation [94] [31], and robotics [4] [111]. This diverse set of domains is a demonstration of the wide variety and impact that CPS techniques have in complex design spaces. Challenges solved in one domain may provide a larger-scale solution for general CPSs. For example, solving control system cybersecurity concerns in the domain of self-driving cars may also apply to nuclear reactors and manufacturing plants, even though these domains do not seem directly related [57].

Thus improving upon CPS design practices, in general, can have a large impact among many engineering domains.

Design of a CPS can be a complicated task due to the interconnected components and real-world interaction [6]. The dynamic behaviors of a system often need to consider important constraints to ensure safe operation. Consider a factory robotic arm, a common engineering practice in CPSs [102]. The robotic arm may need to operate in such an environment where all target setpoints are within a safe boundary. Software for the robot can automatically compute the necessary trajectories for the robot to take to reach each setpoint [108]. Though the setpoints are within a safe region, the dynamically computed path under which the robot moves may enter an unsafe region, causing damage during operation [88].

A method to aid in the design of complex systems like CPSs is to make use of Model-Based Engineering (MBE). Instead of focussing on low-level implementation details, design of a system occurs at a high abstraction level. Since the particular target domain can vary, even within a CPS, modeling can be done within a particular construct, known as Domain-Specific Modeling (DSM). In the case of electrical design, focus can be shifted from discrete components like resistors, capacitors, and transistors, to larger assemblies like logic gates and op-amps. In the case of software, focus can be shifted from functions and primitive data types to objects and messages. Proper domain abstraction ensures that models properly represent the subassembly needed to achieve the model performance. For a CPS, abstractions like electrical op-amps and software objects may still be too low-level for the particular domain, and even further abstractions are needed for a manageable design. Design of a CPS generally has more of a focus on network topology, system blocks, or state diagrams. As similar to the lower-level abstractions, a proper high-level abstraction means that there is a way to transform the high-level models into lower-level artifacts. This lets an engineer focus on model composition at the design phase rather than implementation, and translations of models into low-level, real-working artifacts are

based on the domain's best-known practices.

A more concrete method of employing MBE is through the establishment of a Domain-Specific Modeling Language (DSML). A DSML is a language that allows models to be designed around particular domain concepts and abstractions. DSML implementation also typically implements methods of model interpretation to automatically translate models into real-world functional artifacts. Such model interpretation makes use of codified expert knowledge in the form of software tools, that generate artifacts using best-known methods and practices. Design of a model using a DSML with interpreters is very similar to writing source code with a compiler. DSML interpreters are often called code generators due to their comparison to compilers. For example, a developer in the C language designs program functionality in lines of code. The intention of such a language is to make the design process much more readable. These lines of code represent the high-level behavior, letting the developer focus on functionality rather than particular machine code. The designer of the C language, an expert in processor machine code, understands how the code should best translate into machine code. The expert can then codify their knowledge into the compiler, a useful tool to generate real working machine code from any well-formed C code. This process can reduce, or even eliminate errors of translating C into machine code.

For even higher-level programming, a DSML abstraction can be designed in such a way that it provides a syntax for models in the form of a visual context. Instead of reading lines of code in a design, pictures are effectively drawn to represent the system. A domain-expert may be able to decipher how low-level designs function and interact, but a drawing of connected components is also easy to convey designs to even non-domain-experts. This aids in model validation [22]. As with any other language, a DSML has both syntax and semantics. The syntax defines the rules of construction for models within the domain. Semantically, models in a DSML have a translatable meaning to real functioning components, i.e. lower-level counterparts.

The high-level approach of DSML based design is an attempt to manage system complexity. CPSs are examples of such complexity, and DSMLs have already made a successful impact on CPS design [120] [90] [9] [52] [13] [109] [44]. Multiple domain considerations for a CPS mean that different modeling techniques may need to be implemented in a cohesive manner. One basic method is to provide different aspects for each domain, but let each domain have interfaces to communicate with other domains [18].

Models may be designed using a DSML by using a software-based construction tool known as a Domain-Specific Modeling Environment (DSME). A DSME takes the definition of a DSML's syntax and enforces constraints on a model's construction [43]. This is known as a correct-by-construction methodology [50]. This means that all models constructed within a DSME can only ever abide by the DSML syntax. Correct-by-construction does not however mean that models will meet requirements. Requirements that state desired behavioral outcomes are known as dynamic constraints. In a typical DSML, dynamic constraints are not modeled since the constraints will not change how low-level artifacts are generated. The model designer is then responsible for ensuring that their model behaves correctly under a set of dynamic requirements.

Artifacts may however be generated not only for a functional solution but also for verification tools. These verification tools effectively simulate the dynamic behavior of a design. The output of verification tools may then be compared against the set of requirements. Should any constraint violation exist, the user could be notified of the behavioral error, so that they could modify their model into a potentially correct revision. Such model correction may also be performed automatically. A designer of a DSML may be already aware of best-known design practices based on particular constraint violations. The designer could capture this knowledge by codifying their knowledge of constraint violation correction methods and algorithmically modify the model. This concept is known as Dynamic Constraint Feedback (DCF).

Design of a DSML with interpreters is already a challenging task, so designing a DSML with DCF adds even more development time. The presented approach to aid in the development process is through the introduction of the Dynamic Constraint Feedback Metamodeling Language (DCFML). The intention of the DCFML is to reduce development time in DSMLs implementing DCF while increasing the utility of produced DSMLs with the inclusion of DCF. This not only aids the DSML developers, but end users of the language can see benefits by being provided expert knowledge during the design of their specific models. Models produced with DCF tools will also be ensured to not violate any dynamic constraints, resulting in safer CPS production.

1.2 Motivation

As computer processors and architectures continue to improve, more computational power is available to compute advanced algorithms in real-time. This increase in algorithm performance provides new functionality to enable new technologies like self-driving cars [71]. This however comes at a great cost of increased complexity, creating difficulty in ensuring that all code dynamically performs as expected. Figure 1.1 shows how Source Lines of Code (SLOC) are increasing over time in both aircraft and spacecraft [55]. The automotive industry also has similar trends in SLOC increase over time [97].

From a technical standpoint, potential for introduced errors increases with complexity. Even the smallest of errors can lead to catastrophic results, such as the unit conversion error in the Mars Climate Orbiter [95]. This particular error was a minor issue between the communication of different components, and was not caught due to being shrouded in system complexity. A similar disaster occurred with the Ariane 5 [70]. A common practice in engineering and especially software is reuse, making use of previously known working systems. The Ariane 5 had used the same flight software as the Ariane 4, however the larger scale of the Araine 5 was not anticipated

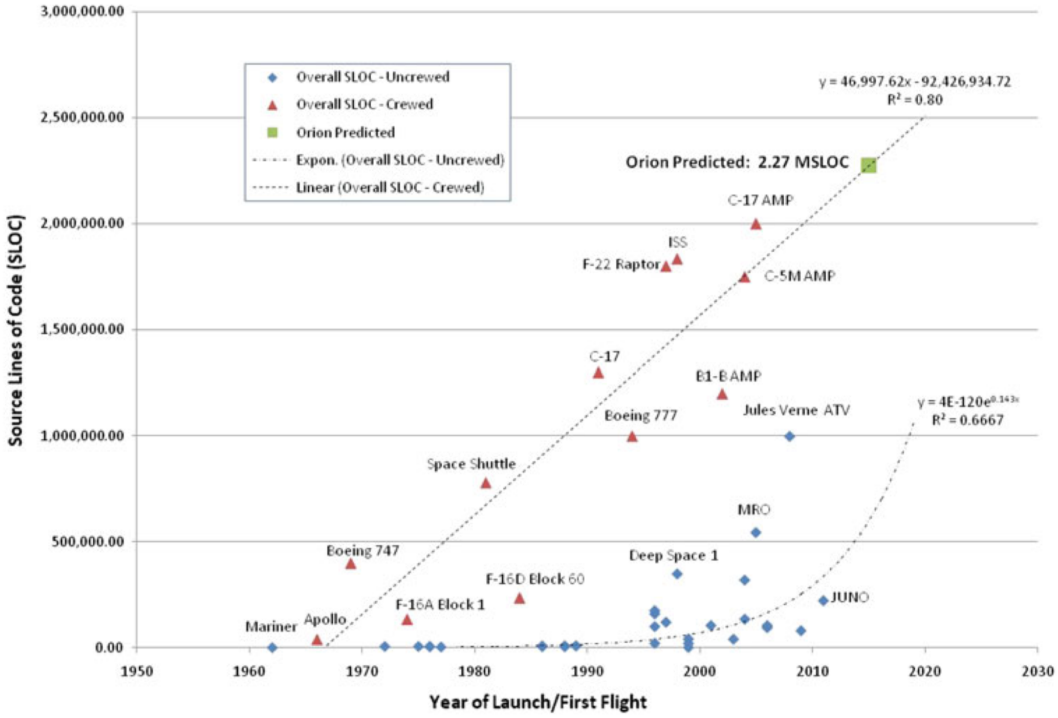

Figure 1.1: Complexity growth shown as increasing Source Lines of Code in aircraft and spacecraft [55].

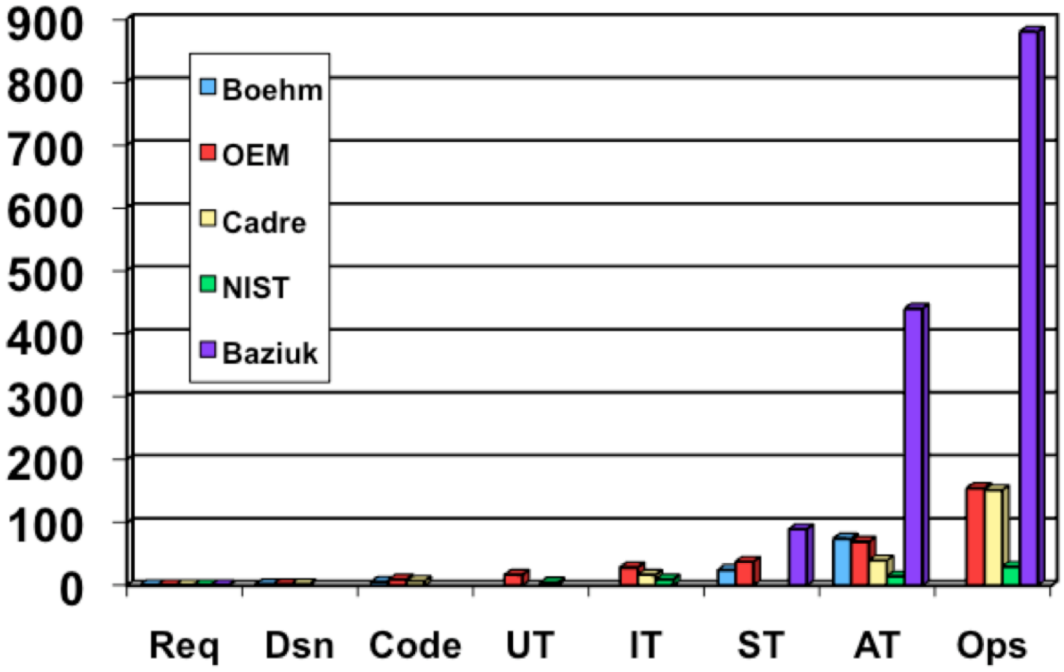

Figure 1.2: Cost of corrections at different stages of development [69]. Definitions are in Table 1.1.

to cause issues. The inertial measurement system was incapable of measuring the larger physical scale of the rocket, resulting in an overflow in part of the feedback for the engine control system.

From an economic standpoint, this exponential increase in complexity also increases the cost of development. Correcting for errors at the end of the development cycle is more costly than at the beginning [92]. A survey of software engineering economics has shown the only 3.5% of errors are found during the requirements and design phase, and 80% are found during or after integration tests [69]. Figure 1.2 shows a breakdown of definitions for Figure 1.2.

There also exists a wide variety of verification tools, often with an overlap in output data. Making use of verification tools is critical to check the dynamic behavior of systems, but choosing the right tool often has barriers. Such barriers include the tool's accessibility and the tool's system requirements. By democratizing verification

Abbreviation	Definition
Req	Requirements
Dsn	Design
Code	Coding
UT	Unit Testing
IT	Software Integration Testing
ST	System Integration Testing
AT	Final Acceptance Tests
Ops	In Service Operations

Table 1.1: The list of definitions for Figure 1.2.

tools, adoption could be greatly increased, resulting in better, verified designs.

1.3 Potential Impact

CPS design is continuing forward as new technologies and design methodologies are being introduced. CPS researchers and developers are finding new ways of designing systems while also using traditional design methods, and understand the importance of verification. This process normally involves a modeler manually designing, simulating, and correcting until verification is passed. This is not scalable as complexity continues to increase, as shown in Figure 1.1. The solution is not to limit the exponential growth in lines of code but account for it with better development techniques.

There has already been much research into the abstraction and analysis of CPSs into DSMLs. DSMLs provide a domain-specific development process paradigm to benefit both the model designers and the resulting target artifacts. Adoption of DSMLs, though a practiced effort, is not particularly prevalent. Increasing further utility can increase such adoption by making DSMLs more attractive than traditional methods. The more useful utilities available to designers, the less time and money spent on developing projects. By defining an architecture for DCF, time to implement automatic model correction methods may be reduced, and resulting

models can be verified for safe behavior amongst a breadth of CPS based domains.

1.4 Contribution

DSML methodologies are becoming more mainstream in CPS development. This work aims to provide improved methods that can improve the quality and utility of a DSML. The presented framework involves a set of novel ideas that build upon the state-of-the-art. The core contribution of this work is to modify the standard metamodelling methodologies to contain integration of DCF upfront in modeling language design. This method may also be expanded to other DSMLs with desired utilities outside of DCF, by shifting focus from a general approach to a more constrained DSML design.

This dissertation is organized as follows. First, a high-level overview of DCF is presented in Chapter 3. This framework is the abstracted practice of design involving verification tools in the loop. The structure provided by this DCF model provides the needed design constraints to build a well-defined DCF based DSML. Chapter 4 takes this abstraction of DCF and presents a concrete design method for DCF based DSMLs. This concrete implementation is called the Dynamic Constraint Feedback Metamodeling Language (DCFML). This language contributes to the state-of-the-art in two factors. The direct contribution is from deployment of the language itself by improving DSMLs for CPS applications. The indirect contribution is through general DSML design, by thinking of structural constraints on the metamodel process to add to DSML utility. Finally, two simple CPS based case studies of DCFML implementation are provided. The first case study is presented in Chapter 5. This case study involves the adaptation of a prior DSML to now fit under the new construct of the DCFML. Since the case study involved a form of modeled dynamic constraints, the necessary metamodel refactoring is also shown. The second case study is presented in Chapter 6. This study involves the development of a new language under the DCFML, with a focus on full automatic model

correction and off-the-shelf verification tools. Different correction methods are also shown to demonstrate how different corrections can result in model evolution behavior akin to traditional control system behavior. Also, this language demonstrates how verification tool input interfaces in similar domains can be modeled differently.

CHAPTER 2

Background

2.1 Model Based Development

Model Based Development (MBD) encompasses the method of design through the use of abstracted domain elements known as models [87]. A design utilizing MBD is a set of domain concepts that are associated and interact with one another. Since such abstractions may be unique to a domain, multiple domains, or even a subset of a domain, MBD is often successful if performed through a DSML. Such domain abstractions may be represented in various forms, including visual, textual, or mathematical representations [28]. Each domain may be quite different, thus specific forms of modeling approaches may be better suited than others for target domains. Each domain may have a different set of semantics and syntax that need to be defined. The concrete syntax of a model is defined through the design of a metamodel. The metamodel defines how models may be constructed, i.e. is a model of the model. A metamodel is responsible for defining the set of allowable domain design components and how they are interrelated. The design of a metamodel is normally performed by a domain expert.

Implementation of a DSML typically benefits from two major utilities: correctness from construction, and model interpretation. If a metamodel is determined to be valid for the domain, then models created by following the metamodel can be considered to be correct-by-construction. This means that the model is syntactically correct. Programming languages and practices provide a similar level of thought when thinking about modeling languages. A programming language necessitates code to be well-formed before tools such as compilers may be run. During

development however, syntax is typically temporarily broken since functions and declarations need to be typed out. In the middle of typing a line of code, the syntax is invalid. Syntax errors become one of the simplest yet most common mistakes in programming. If however a line of code is copied and pasted, then the code will be syntactically correct before and immediately after the change. This is effectively modifying the code by implementing something that is guaranteed to follow the syntax, and can be considered to be correct-by-construction. In the case of a custom metamodel, typically domain concepts are generated from the formal definition. This is typically done through the use of a DSME, to be discussed further in section 2.1.1.

The second major tool that greatly increases the utility of a DSML is the use of interpreters. Again, programming languages have a similar utility in the form of a compiler. The use of a compiler removes the need to translate the code by hand into another language. Converting C code into machine code can be a daunting task, and can be even more daunting with higher-level languages like C++. A well-trained person may also make simple mistakes during the translation process, adding to the development time. The use of a compiler is a model interpreter that has the functionality to automatically translate the program model into a working set of artifacts. An expert in the domain of both the C language and in machine code can algorithmically capture their expert translation knowledge in the form of the compiler. This process can be similarly accomplished in the scope of general MBD, by letting the metamodel designer create interpreters that can iterate over models and produce domain artifacts. In terms of a DSML, the particular domain may encompass a wide variety of necessary artifacts. For a CPS, this can involve computation algorithms (source code), control systems (schematics or code), and networks (hardware configurations). Some specific modeling languages bridge the gap between such disciplines, such as INTO-CPS [66]. It may be expected that an interpreter for a CPS model produces software, controller tunings, or schematics, or

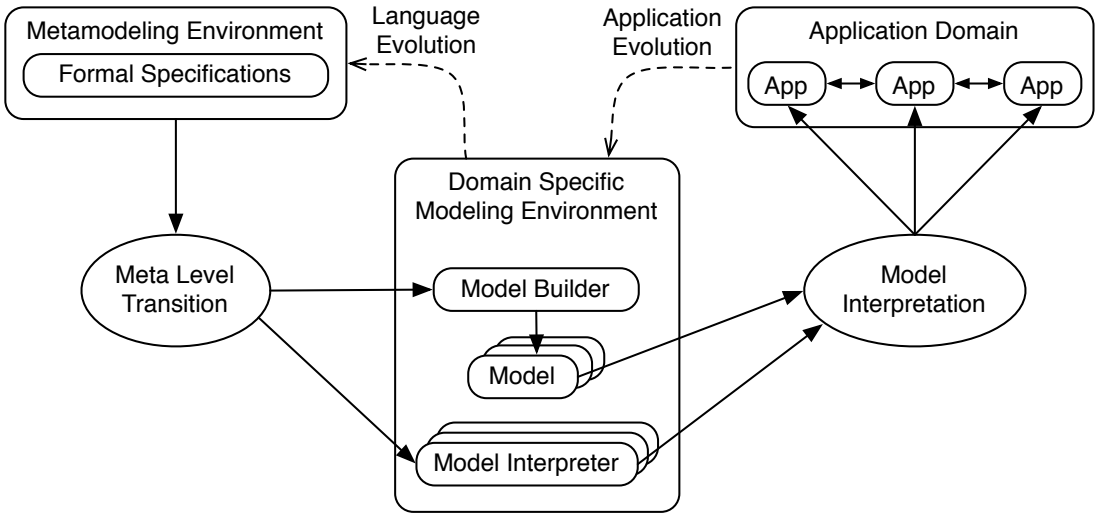

Figure 2.1: The components of Model Based Development.

network configuration information. Interpretation is a form of model transformation, discussed further in section 2.1.3.

Figure 2.1 shows how models, metamodeling, output artifacts, and interpreters fit within the context of general DSML design. This diagram can be related to development of source code by observing the *Model* and *Model Interpreter* as similar to source code and compilation, respectively. Compiling code produces target artifacts in the *Application Domain*, through *Model Interpretation*. *Application Evolution* is then the process of trying out code by executing the application, then correct the program based on observed errors. A DSML works with a DSME to provide further tools, such as the *Model Builder* to provide a correct-by-construction methodology to build models. A DSML is defined by *Formal Specifications* in the *Metamodeling Environment*. Generally, the *Metamodeling Environment* has a special interpreter known as the *Meta Level Transition* to generate the rules for the *Model Builder*, as well as templates for *Model Interpreters*. As similar with general code development, if the domain abstractions are incomplete or incorrect, then the metamodel may be refined through *Language Evolution*.

Development of a DSML may begin at any of the abstraction levels. In the case of previous work in a field rich with applications, a DSML design begins by abstracting the applications into modeling techniques, and eventually into a metamodel. In a brand new domain, models may be drawn first for a particular aesthetic, followed by interpreters for practical use. Starting with a working set of code is possible to more easily build the set of interpreters by converting working code into templates [115].

Methods have been previously done in the formalization of DSMLs [49] [37] [27]. Such works have established a formalization of a language L as a 5-tuple as shown in Equation 2.1.

$$L =< C, A, S, M_S, M_C > \tag{2.1}$$

Where C represents the *Concrete Syntax*, A represents the *Abstract Syntax*, S represents the *Semantics*, M_S represents the *Semantic Mapping*, and M_C represents the *Syntactic Mapping*. The *Concrete Syntax* represents the specific notation for model constructs. *Abstract Syntax* represents the modeling concepts and relationships, where the *Syntactic Mapping* relates syntactic notations to elements in the *Concrete Syntax*. The *Semantic Domain* is the set of meanings and behaviors of language constructs, with the *Semantic Mapping* relating the *Abstract Syntax* to elements in the *Semantic Domain*. With these five components in place, a model can then be a realization of a language L.

2.1.1 DSME

A DSME provides the necessary tools to define the meta specification and to build models based on the formal specification [32]. DSMEs are the technical implementation of the theory of Model Integrated Computing (MIC), by providing the tools to aid in metamodeling, model building, and interpretation. Figure 2.1 shows how a DSME provides the framework to build metamodels, then uses the specifications to provide a model builder and interpreter integration. The model builder is only

allowed to create models that abide by the metamodel, thus the DSME aids in providing the correct-by-construction methodology. Usually a DSME provides the metamodeling environment in tandem with the model builder. By providing the tools that provide the key utilities for MBD, modeling languages can be rapid prototyped. The use of a DSME can greatly aid in the success of designing a CPS [34].

Various DSMEs exist, including GME [67], MetaEdit [101], GEMS [113], Eclipse Modeling Project [103], and OMG [16]. Some DSMEs even focus on functionality for particular domains, such as the CPS focused OpenMETA [104] [105]. All of these DSMEs provide a meta-level definition suite for metamodel design, and each allows for different process requirements, interfaces, and modeling methods. Some provide a more visual context to the metamodel design, whereas others define the metamodel in an XML schema.

2.1.2 Web-Based Generic Modeling Environment

The Web-based Generic Modeling Environment (WebGME) [76] is an example of a DSME that provides a method of designing a metamodel that closely represents a class diagram from the Unified Modeling Language (UML). This particular environment is well suited for software experts with experience in UML and Javascript. Design of language syntax is fairly easy due to model visualizations, enabling rapid prototyping of the formal specification. Languages are designed to be mainly visual, where models are typically represented as graph-like connected nodes and edges. The prototyping phase of a language is generally a quick process due to the drag-and-drop nature of the metamodel design interface.

WebGME also features template generation for decorators, add-ons, and plugins. The Javascript-based plugins using WebGME's Core API are invokable through the web interface and provide model traversal and modification methods. The Core API is the effective method of designing model interpreters and model transformations.

WebGME provides a method of decorators following the software-based decorator design pattern, to aid in improving language quality through customizable aspects and visualizations of models during creation. A decorator may be used in the aid of more closely matching the visual representation in a design domain, i.e. drawing correctly shaped boxes based on the UML standard. Custom visualizations may also take advantage of a decorator's use by providing verification results directly on the model.

WebGME is not necessarily CPS focused but is generic to capture a superset of design domains. WebGME has been used for autonomous vehicle applications involving verification [79]. The WebGME-based SURE language is targeted towards security and resilience evaluation in CPSs [85]. A taxonomy of verification tools resource is also making use of WebGME for verification tool evaluation in CPS applications [56].

2.1.3 Model Transformations

Model transformations encompass the design of functions that take an inputted set of source models and produce a set of target models. Source and target models may exist at different abstraction layers and with either different or similar meta-model definitions [81] [29] [30]. An interpreter or compiler, for example, transforms the model from a high-level language into a set of low -evel artifacts. Conversely, rewriting a program written in C++ into Java would be at the same abstraction level but under a different formal specification. Model transformations may also exist as model evolution by modifying the functional graph, maintaining the meta-model between the source and target models by only needing to modify the model structure.

Some frameworks have been written to handle a variety of transformation methods such as ATL for OMG [54] [53] [16]. GReAT is an example that works with the Generic Modeling Environment, the prior version of WebGME [67] [1] [14]. Such

transformation tools work directly with the metamodel definitions to easily provide a methodology to define model transformations in a DSME.

The concept of a model transformation is a very simple idea that covers a breadth of converting a model into a different, new model. A taxonomy of model transformations has been established to aid in the recognition of the various qualities different transformations a model may undergo [81]. For example, part of this taxonomy defines a classification between exogenous and endogenous transformations.

- **Endogenous Transformation** - A transformed model will be within the same language.
- **Exogenous Transformation** - A transformed model will be in a different language.

Another, orthogonal classification of model transformations is defining them as either being horizontal or vertical [25]. Further classifications at various levels exist, but only these two categories will be discussed in the context of the DCFML.

Horizontal Transformations

A horizontal transformation occurs when the source and target models both represent the same abstraction level. One form of a horizontal transformation is converting a model from one similarly abstracted language into another language, i.e. language translation. For example, converting a program written in C++ to Java would both represent a new language but at a similar abstraction. Another term for this transformation would be language migration, which is considered to be an exogenous horizontal transformation. In this particular work, horizontal transformations are used in two parts of the DCFML:

- To establish a standard interface between multiple verification tools.

- To translate dynamic constraint violations into expert block inputs for model correction.

The other form of a horizontal transformation is an endogenous transformation. In this case, the language stays the same and is under the same abstraction level. An example of this transformation is also known as model evolution, where the model is modified but is within the same language. Such horizontal transformations may involve tuning of a value or restructuring of elements. Horizontal endogenous transformations happen during the design phase of a project, where high-level models undergo transformation remaining at the same abstraction level. In software development, refactoring is a common example of this classification. For the DCFML, horizontal endogenous transformations are used in the following:

1. To be performed by the expert blocks, as a model correction step.

Vertical Transformations

A vertical transformation represents a different abstraction level between the source and target models. Since the abstraction level is different, it is easier to think about vertical exogenous transformations. Here, both the abstraction level and language are different. The simplest example of this is a software compiler, where a higher-level language is transformed into target artifacts that can be executed on a hardware platform. For a DSML in general, an interpreter is generally built to provide what is commonly known as a code generator, providing a great amount of utility to the language. For the DCFML, vertical exogenous transformations are used in three areas:

1. As a final code generation method, defined by the DCFML user as similar to designing a typical DSML, to generate real working code from their DSML.

2. To translate the model into the necessary inputs for verification tools.

3. As a provided DCFML interpreter, converting the model of DCF into the plugins necessary for the user's DSML to integrate their DCF definitions.

A vertical endogenous transformation involves the same language, but at a different abstraction level. Another term for this is formal refinement, which is a common development step in the iterative process of DSML design. If a model is created and there is an incorrect or incomplete syntax or semantic mapping, then the metamodel needs to be refined. This is not a step that is done when developing in languages like C++, since the language cannot be modified. The DCFML has employed vertical endogenous transformations as follows:

1. The DCFML has provided a metamodel builder for a DSML user, to evolve their metamodel.

2. The DCFML, in its current form, is the result of a vertical endogenous transformation by adding structure and constraints to WebGME's base meta-metamodel.

2.1.4 Software Design Patterns

Software design efficacy greatly increases when implementing well-known design methods known as software design patterns [40]. Expert knowledge may be codified through the abstraction of solutions to common software problems [118]. Thus, with the implementation of software patterns, software design becomes less error-prone and readable.

For example, if a developer is to write a program that requires multiple aspects of a program to access a single resource, such as navigation application where the UI displays current whereabouts while a background algorithm needs to constantly poll GPS information, then two separately programmed components may conflict when accessing the same resource. A beginner developer may attempt to implement an effective-yet-complex logic solution to ensure mutual exclusion, treating

the problem as a new issue. A more well-versed developer may recognize that the Singleton design method is used to ensure that a single object is used amongst multiple class instantiations. If a new developer were to read the code from both, the singleton design pattern will be easy to recognize and understand, whereas the beginner developer's solution will require in-depth reading to understand the logic involved.

The use of well-known patterns in software development also results in the focus on higher-level modeling concepts rather than low-level implementation details [58]. Such design at a higher-level ensures easier traceability to requirements and thus aids in the validation of software design.

2.2 Cyber Physical Systems

2.2.1 Autonomous Vehicles

Autonomous vehicles (i.e. self-driving cars) are examples of a CPS, usually implementing all three core subdomains of a CPS (networking, computation, and control). Research in this area received a major push after DARPA announced the original DARPA Grand Challenge in 2004 [20]. No group was able to complete the challenge until the challenge was offered a second time [107]. In 2019, there are many companies and research groups developing autonomous vehicles [10]. Some groups are developing model based design methods of automotive vehicles [33]. One of the case studies of the DCFML presented here is based on a deployed language that functions on the CAT Vehicle, and is based on a prior trajectory-based modeling language [79].

2.2.2 MATLAB and Robotics Operating System

The Robotics Operating System (ROS), unlike the name suggests, is not an operating system but rather a middleware [91]. ROS is targeted for CPSs, particularly

for distributed computation applications. Being a middleware, ROS provides a set of hardware and network abstractions, providing computational cluster capabilities through message passing structures. This framework aids complex system development by simplifying the interfaces and providing feature-rich utilities including data recording and the Gazebo physics simulator [60]. The paradigm of design is shifted to be focused on component based design rather than needing to juggle network code with a target domain. Part of the component design also allows for effortless Software-In-The-Loop (SWIL) to Hardware-In-The-Loop (HWIL) testing.

The popularity of ROS has caught the attention of other high profile software suites, including MATLAB. MATLAB's Simulink, a DSML for Matlab, includes code generation tools that produce C++ artifacts [19]. Simulink is capable of generating ROS code [80]. This particular framework is interesting due to its high flexibility, verification potential (MATLAB tool suite), and its ability to generate HWIL/SWIL code. Using such a framework would be a prime candidate for the DCFML, however Simulink's metamodel is proprietary and therefore the loop cannot be closed for automatic DCF.

2.2.3 Verification Tools

A verification tool is generally a specific piece of software that aims to simulate models using particular methods. Most of what fits under this classification may not even be labeled as a verification tool, but is rather classified as a method of checking qualities of a model in a particular domain. Thus finding verification tools can be a challenging task without having unified, searchable metadata. While many researchers aim to find solutions for particular problems, some find the need to develop novel methods to check their efforts. The resulting checking methods are then sometimes scaled up for use by others, and become known as a verification tool. Sometimes efforts are done on model checking without providing a specific tool, such as checking Linear Temporal Logic (LTL) [106] or some CPS model checking efforts

[42].

Some examples of verification tools include verifying LTL (SPIN [15]), Hybrid Systems (SpaceEx [39], CORA [3], FLOW* [23], dReach [62], HyCreate [12]), level set methods (Level Set Methods Toolbox [82]), digital systems (DSVerifier [48]), bisimulation (CADP [41]), falsification (S-Taliro [5]), and real-time systems (UPPAAL [65]), to name a few. Though many appear to have direct overlap by verifying the same domain, some tools provide different qualities of the solution from implementation of different computational methods. Some of these even result in producing information that may be of direct use for model correction, such as dReach being able to provide a hybrid system's proper initial conditions to reach a particular state. Some other reachability tools like HyCreate do not provide such information.

One of the active efforts in organizing all verification tools for CPS applications is the Cyber-Physical System Virtual Organization (CPS-VO) Active Resources [93]. A portion of this effort is to create a functional taxonomy of verification tools. Part of the effort is to provide examples of use, and eventually virtualization to remove the requirement of particular system specifications to execute tools. A follow-up effort to this is creating design studios for verification tools for rapid evaluation [56].

2.3 Control Systems

Control system design theory is based on the principle of providing inputs to a dynamical system to achieve a new, desired set of dynamics. The dynamic system to be under control is known as a plant, consisting of a process and actuation. Generally, the plant does not exhibit the desired behavior. For example, a person sitting in a car with an automatic transmission may want their car to drive at a particular speed, but the car is unable to perform this with no control input. A throttle is provided for the person to modify the engine's power which eventually modifies the speed state of the car. There are many different control methodologies that may be suitable for different plants. One of the simplest controllers is a feedforward con-

troller. In this case, inputs are provided to a plant, and the result is not observed by the controller. This would be akin to driving a car with a blindfold. In principle, it is possible to drive to a location from memorizing the dynamics of the car and the roads, but any disturbance or error in understanding the car's dynamics can be catastrophic. Removing the blindfold allows the driver to adjust by sensing the surroundings and adjust for such errors, in what is commonly known as a form of feedback control.

2.3.1 Linear Time Invariant Systems

A Linear Time Invariant (LTI) system is a system where the output is linearly related to the input and has no dynamic variation in time. For example, consider a plant $g()$ for the system, with output $y(t)$, and input $x(t)$ as in Equation 2.2.

$$y(t) = g(x(t)) \tag{2.2}$$

If a system satisfies the equality in Equation 2.3, then it is said that the system is linear, since it obeys the superposition principle.

$$g(a_1x_1(t) + a_2x_2(t)) = a_1g(x_1(t)) + a_2g(x_2(t)) \tag{2.3}$$

Secondly, a system is said to be time invariant if the response of the system is not dependent on time. This means that if the same input is provided, regardless of a shift in time, the same output will be produced. This concept is described in Equation 2.4.

$$y(t) = g(x(t)) \rightarrow y(t + \Delta) = g(x(t + \Delta)) \tag{2.4}$$

An LTI system is then a valid candidate for a simple transformation from the time domain into the complex frequency domain, as defined by the Laplace transform. This seems like a strange jump and questions of effectiveness arise, however

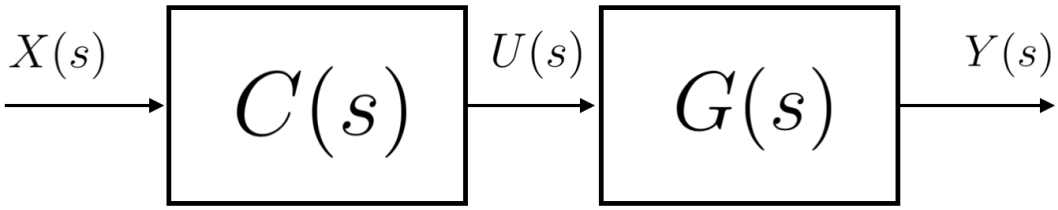

Figure 2.2: An example system in the complex frequency domain.

this transfer opens up a new domain of analysis and design possibilities. Performing a Laplace transform can take a calculus level problem in the time domain to produce an algebraic level problem in the frequency domain. The Laplace transform definition, to convert a signal $g(t)$ into the complex frequency domain, is shown in Equations 2.5.

$$G(s) = \mathscr{L}\{g(t)\} = \int_0^\infty g(t)e^{-st}dt \tag{2.5}$$

Similar to converting calculus problems to algebraic problems, a Laplace transform often eases system design by making system design easier by allowing for easier separation of system blocks. This concept involves thinking about the concept of transfer functions. When one signal input is provided to the input of a function, time domain analysis needs to make use of convolution to determine the output signal. The system of convolution, when converted into the complex frequency domain, becomes simplified into a multiplication. Figure 2.2 shows a common representation of a system in the complex frequency domain. The transfer function for $G(s)$ is defined as in equation 2.6.

$$U(s) = G(s)X(s) \implies G(s) = \frac{U(s)}{X(s)} \tag{2.6}$$

The full system transfer function can be determined as follows.

$$C(s)G(s) = \frac{Y(s)}{U(s)}\frac{U(s)}{X(s)} = \frac{Y(s)}{X(s)} \tag{2.7}$$

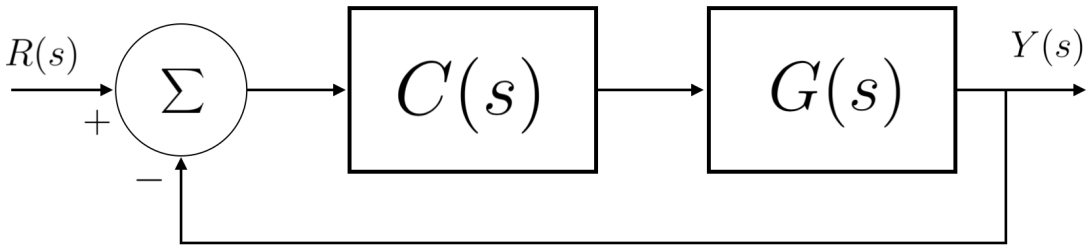

Figure 2.3: A closed-loop feedback constol system.

Figure 2.2 is representative of a feedforward control system of plant $G(s)$ with controller $C(s)$. As stated before, feedforward control is effectively blind and is highly dependent on model error and initial conditions. An example of a simple feedback control mechanism is shown in Figure 2.3. In this case, the output is measured and compared against a reference input, producing an error for a differently-designed controller $C(s)$ and in the feedforward case. The full system transfer function for feedback control is shown in Equation 2.8

$$\frac{Y(s)}{R(s)} = \frac{C(s)G(s)}{1 + C(s)G(s)} \tag{2.8}$$

Second Order Systems

A second order system is a subset of linear system models, characterized by having two poles. A second order system is often a simplified yet acceptable approximation of many real world dynamics. For example, often a physical dynamics example is constructed using a mass, a spring, and a dampener. In electronics, a system that contains two successive filters also falls under second order system modeling. Generically, a 2-pole system may be described as a transfer function in equation 2.9.

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{2.9}$$

This transfer function is constructed in such a way that basic time domain re-

sponse characteristics can be directly observed. The characteristics may be adjusted with both the natural frequency ω_n and dampening ratio ζ. By adjusting the values for ω_n and ζ, the time domain properties change. Often one of the first questions of a designed system is how quickly the system approaches its equilibrium point. There may be varying definitions of what is considered to be "close enough", but often it is seen as reaching within 2% of the equilibrium from a step response. For this case, the settling time t_s solution is shown in Equation 2.10.

$$t_s = \frac{-\ln\left(0.02\sqrt{1-\zeta^2}\right)}{\zeta\omega_n} \approx \frac{4}{\zeta\omega_n} \tag{2.10}$$

A corollary to this is analyzing the transient response characteristics, specifically how quickly the system approaches the equilibrium from a step input. Again this may be defined in various ways. Typically for an over dampened system, this is defined as the time to go from 10% to 90% of the final value. The rise time t_r for this scenario can be determined in Equation 2.11.

$$t_r = \frac{2.16\zeta + 0.60}{\omega_n} \tag{2.11}$$

In cases where the dampening is weak, the system exhibits oscillatory behavior, going beyond the equilibrium point. This behavior is known as overshoot. In some cases, it may be desirable to prefer rise time performance over overshoot performance. However, there are some safety-critical systems where an overshoot may drive a physical system into a boundary. Thus in the case of overshoot, the amount of overshoot may be of importance to determine. Equation 2.12 shows the calculation for the percentage of overshoot PO. Notice that the overshoot is independent of ω_n, and the overshoot equation is only valid, i.e. overshoot only exists, when $\zeta < 1$.

$$PO = 100 * e^{\left(\frac{-\zeta\pi}{\sqrt{1-\zeta^2}}\right)} \tag{2.12}$$

A corollary to the existence of overshoot is determining the time the peak value occurs. This peak time t_p can be calculated, again if $\zeta < 1$, with Equation 2.13.

$$t_p = \frac{\pi}{\omega_n \sqrt{1 - \zeta^2}} \tag{2.13}$$

2.3.2 Hybrid Systems

A hybrid system is a system involving both continuous and discrete events. This is often the case in CPS modeling and design, due to the generally continuous physical nature yet having the discrete computation [6].

Often in complex systems like CPSs, simple controllers are not sufficient to satisfy complex requirements. Also, not every plant provides a linear method of interaction. As an example, a common home thermostat does not adjust the level of cooling or heating, but only turns an air conditioner or heater either on or off. In other systems like automotive cruise control, traditional control theory can be used to some extent but different control modes are needed for braking and throttle. One method of making use of traditional control theory but attempting to satisfy important safety constraints is hybrid control design [74]. Hybrid controllers change their control modes based on particular conditions, thus combining state diagram design with control design. Different tools are available to simulate a hybrid control design but vary in their methods of evaluation, such examples are dReach [62] and HyCreate [12]. Due to the extensive research being done on reachability and its importance to CPSs, this serves as a great case study for the use of the DCFML. Design of hybrid controllers may even be reduced to simpler, easier to design modes yet still need to consider critical safety constraints [79] [21]. Online resources are also being built to create a taxonomy of verification tools for easier discovery and tool selection [86]. Such online resources include examples for implementation methods, potentially in the future example modeling languages and metamodels could be provided for easier DSML integration.

2.4 Related Work

2.4.1 Automatic Controller Tuning

Control systems are prevalent in CPSs at varying levels of complexity, from low-level components like voltage regulation [35] to high-level integrations like lane control for autonomous vehicles [46]. Many controller types and tuning methods exist for various scenarios [100]. Many systems to be controlled may be approximated as second order systems, however model errors result in varying controller performance. Plenty of efforts exist to automatically tune controllers for systems where even approximations are difficult to establish. The PID controller is an example well-rounded control method that is widely used in industry [110]. PID stands for three gains with processes computed on the error signal: Proportional, Integral, and Derivative. Each of these three methods of handling error are coupled into a single controller providing different dynamic characteristics. Because of the prevalence in real-world systems, methods for determining the three optimal gain values in different scenarios are often researched [96] [8] [75]. One such method to tune controllers based on a black-box view of a system plant is known as the Ziegler-Nichols method [7]. This method is a heuristic-based algorithm that attempts to achieve a particular quality of tuning based on a general analysis. This method involves adjust one gain until particular qualities are observed from testing the system, then further gains are adjusted until the algorithm is complete.

2.4.2 Dynamic Constraint Feedback

One of the natural efforts in designing a model interpreter is to run a model in a real system to check the system dynamics. This leads to an evaluation of the system where the syntax alone cannot check that dynamic constraints have been met. Interpreters may also be written to translate the model into artifacts for verification and simulation tools for DCF. Past efforts have been conducted to use DCF in

particular custom modeling languages [116] [114]. These works encompass real-world engineering problems with custom made codified expert knowledge to automatically correct the model. This includes controller tuning through value modification or adding new gains to an existing controller. Other efforts have implemented DCF using very different DSMLs, such as concurrent state diagram modeling using the SPIN model checker [119]. In this example work, concurrent state machines issues such as deadlock are corrected by adding states or transitions.

The domain of Computer Aided Design (CAD) has recently made use of a form of DCF in the form of generative design [63] [72] [98]. Generative design methods exist in professional CAD software such as Solidworks. In this particular case, a set of criteria is provided to an initial model, and the generative process either adds to or cuts away parts of the design while checking that the design criteria have not been broken. In other words, models are modified based on the set of constraints, and therefore may be thought of as a DCF example. This is similar in other CAD applications, such as envelope design based on assembly criteria [38].

The domain of Printed Circuit Board (PCB) design involves tools that sometimes provide a feature called autorouter [84]. PCB routing involves connecting electrical components with traces based on an electrical specification, but simultaneously needs to account for manufacturability constraints. A PCB design tool usually provides a verification tool to ensure that traces of separate electrical connections do not cross and that all connections have been made. In general, autorouters attempt to provide a fully automated process by incrementally modifying the design to avoid constraint violations. Some autorouters also provide partial automation [45].

Notable in these prior other works with DCF based model correction is that a high-level perspective is used when describing the methodology, but each example's implementation is performed at a low level in the modeling language design. The focus in each work is on the methods used to close the loop on model correction. Each describes the definitions of how models are created and how they should be

modified based on the output of specific verification tools. The data types at each stage vary greatly, though the high-level discussion about the design is very similar in each work. It is these past works that serve as one of the focuses for the DCFML design presented in this work.

2.5 Problem Statement

Design of CPSs has shown a trend in the implementation of DSMLs for better design management. Recent tools and methodologies have implemented forms of DCF methodologies, however these have been realized on a case by case basis. The engineering design process involves the implementation of known solutions to meet requirements. In more complex systems, high-level modeling is used before transitioning to a low-level implementation. This work seeks to resolve and discuss these issues by developing and exploring the following tasks:

- Develop an abstraction of DCF for modeling language design.
- Develop a concrete, democratizing framework for DCF based DSML design.
- Ensure the framework closes the loop on automatic model correction.
- Ensure off-the-shelf tools can be implemented in the framework.
- Ensure dynamic constraints can be modeled by the framework.

CHAPTER 3

A Verification Feedback Framework

3.1 Components in the Framework

Section 1.1 has already provided a brief overview of closing the loop on DCF, but only as a high level thought experiment. Figure 3.1 is an example high-level visual representation of a modeling language that employs DCF. This figure shows how the implementation of DCF may be thought of as a component based design, with blocks representing Model Transformations, Verification, and Constraint/Behavior Comparison. Lacking in the diagram are low-level details, and most importantly how different components can interact. The model transformation block needs to work with the model, i.e. model transformations need to be designed along with a metamodel. Similarly, a designer of a metamodel needs to be aware of both the deployment interpreter and the inputs necessary for verification tools. This section will serve to dive a bit deeper into the definitions and concerns of establishing a DCF framework by looking at these individual components. The deployment interpreter will not be discussed in great detail since it is a common part of DSML design and is no different in this implementation. This section will also consider some of the challenges of implementing DCF, and how the framework can serve to benefit a DSML designer when either creating a new language from the ground up or adapting a prior language to implement DCF.

Figure 3.1 is a visual aid to the concept of closing the loop on DCF. An initial model is first provided as x, which may be directly deployed or may be interpreted into artifacts necessary for verification tools. Execution of a verification tool provides the set of dynamic behaviors b, which may be compared against the set of

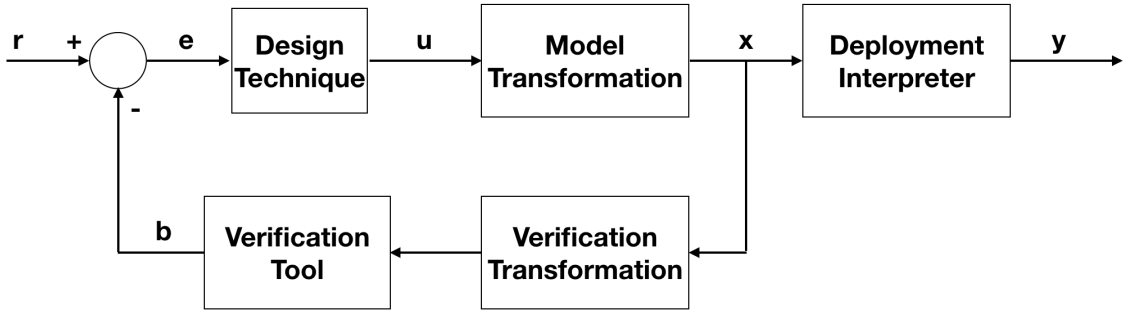

Figure 3.1: A closed-loop dynamic constraint feedback modeling language.

requirements r. Should any constraint violations e exist, an expert block may decide on the appropriate model correction method. With a known model modification method u, an update to the model may be performed. This cycle may be performed if necessary multiple times until all constraint violations are removed.

3.1.1 Metamodel

The language syntax is defined by the metamodel and enforces structural constraints in a DSME. The variable x in Figure 3.1 represents the model created from the metamodel. The metamodel is not shown in Figure 3.1, but is necessary to construct x. As with any metamodel design, a semantic mapping must exist between the metamodel structure and the target domain. Employing the DCF has an added further metamodel design constraint, since it also needs to have a semantic mapping to verification tools. If a metamodeler were to design a DSML with DCF and had no prior knowledge of a DCF framework, then the metamodel would also need to have a model of the dynamic constraints. In Figure 3.1 however, the set of dynamic constraints are represented as requirements r, providing a more structured approach to dynamic constraint modeling. This prevents an unrestricted implementation of building a metamodel, providing the following benefits.

- Allows a DSML designer to build metamodels with well-established design methods, without the burden of complicating the metamodel with dynamic

constraint modeling.

- Keeps a clear separation of dynamic constraints versus structural constraints.
- Provides traceability of verification methods and requirements.

Different DSMEs may necessitate particular design methodologies. Some metamodels are defined by UML while others may be defined using XML. These differences should have no impact on this framework, especially since the metamodel does not need to be designed differently. The only added set of constraints is that model evolution transformations need to be designed and implemented, and that the model can undergo transformation for verification tools. These constraints are certainly not small considerations, however if met they do provide the potential benefits of a fully automated DCF model correction.

Design of metamodels under this framework needs to consider the verification tools for the requirements. When analyzing the inputs to a verification tool, one of the first methods may be too abstract the inputs into a verification tool metamodel. The metamodel should be domain specific, however this does not mean that they need to be verification tool-specific. For example, if the design domain were for a vehicle controller, it may be assumed that the vehicle will never change. Similarly, a closed loop controller may be expected as part of a metamodel. This means that only the controller needs to be defined, but parts like the vehicle plant and the feedback mechanism are implicitly defined. A verification tool may be able to define a full set of signals and systems, but an implicit framework can already be defined by the metamodeler. This greatly eases the constraints of implementing verification tools in the metamodel, but only needing to consider domain-specific elements. Avoiding full verification tool abstraction also benefits the use of multiple verification tools that may be difficult to combine.

Similarly, creating a highly domain-specific language lets the metamodeler focus on the implicit characteristics of design. The model transformations are easier to

define is the language is simpler, or has specific elements targeted for model evolution. If a metamodel were to be too generic, then the model transformations would be difficult to target particular, acceptable parts of the model for modification. For example, if again the design domain were for a controller for a particular car, then distinguishing between the plant and controller in a signals and systems generic model may be non-trivial. Thus again, the more specificity in the domain, the easier the implementation of DCF.

3.1.2 Interpreters

The interpreters serve as the glue logic to interface the model to both verification tools and real-world deployment. These exist as vertical exogenous transformations, transforming the model in the DSML to lower level target artifacts. In the case of the deployment interpreter, this has been standard practice in language design ever since the first compiler was built. The deployment interpreter transforms the model x into the real working output artifacts y. Implementing DCF has no particular restrictions on the deployment interpreter, however if a prior language was modified to employ DCF, then language differences may need to be accounted for. There may be two methods in this approach:

- Restructure the prior deployment interpreter to account for new modeling differences (i.e. a new concrete syntax or semantic mapping).
- Design a new horizontal exogenous transformation to convert the new verifiable model into a model of the old metamodel, for prior deployment interpreter use.

Again, this is only in the case that a prior language needs to be adapted for DCF use, whereas a new language will undergo the traditional steps of deployment interpreter development.

The second use of an interpreter occurs during the verification steps, by transforming the model into verification inputs. This interpreter is usually an additional

development step for a DSML, whether the language is new or old. A rare case may also exist where the deployment interpreter produces artifacts that identically match the inputs necessary for verification tools. An example of this instance is the generation of ROS nodes through the design of Simulink with code generation. The set of ROS nodes may be deployed on a full hardware-in-the-loop (HWIL) system as generated, or the same ROS nodes may run through ROS's gazebo simulator in a software-in-the-loop (SWIL) setting and generate rosbag files for dynamic verification. Generally speaking, the verification interpreter will need to be different since most verification tools require a very particularly defined input and quite simply are not acceptable artifacts for deployment.

3.1.3 Verification

The verification block represents a mapping from a model to a set of dynamic behaviors of interest. These blocks represent a set of necessary verification methods needed to satisfy the set of requirements r, by producing comparable behaviors b. This means that the dimensionality and data types of the set b need to match that of the dynamic requirements r. In some simple cases, a single verification tool may provide the exact size of b to match r. Usually, a verification tool provides even more data than necessary, so only a subset of the verification tool's output is needed to construct b. In more complex systems spanning multiple domains, like a CPS, multiple verification tools need to combine their parts of their outputs to produce b.

The chosen verification tools may be custom constructed or from a selection of off-the-shelf tools. In many cases involving a complex dynamical system, well-established off-the-shell tools are the easiest and quickest tools to implement. This is a preferable option for complex verification needs, such as needing a full physics simulation. Using such tools may however still require significant development time regarding the verification tool interpreter. In other cases, no verification tool may

exist for a particular domain or set of requirements. This scenario is usually useful for simple requirements where no verification tool exists, and simple operators are sufficient for determining the dynamic behavior. For example, if a robot was commanded to move between two goal points, an off-the-shelf tool involving reachability may be great for ensuring that no collision occurs between valid goal points, whereas a custom tool would be sufficient for determining whether the goal points are valid in the first place since the data types may be trivial comparisons against requirements.

3.1.4 Constraint Comparator

The constraint comparator as shown in Figure 3.1 is a highly simplistic view of the comparison operation. This symbol is used as inspired from control system theory, but does not represent the true internal complexity of constraint comparison. The constraint comparator will have varying amounts of complexity depending on the design domain and selected verification tools. The comparator is responsible for taking the current dynamic behaviors b of the model, and compare them against the set of requirements r, and produces a set of constraint violations e. In control system theory, e represents the error, where the negative feedback mechanism and controllers try to reduce the error to 0 with particular performance characteristics. In DCF, e represents a set of dynamic constraint violations, with the goal of re-designing the model to achieve an empty set for e, thus passing requirements verification. The constraint comparator is constructed in conjunction with the verification outputs, as defined by the requirement. Normally a requirement is defined in prose, sometimes with a particular associated value. For example:

- "The timer shall not extend past 10 seconds."
- "The robot shall not collide with any wall."

In both of these example requirements, the comparison operation and the corre-

sponding values are described. In the first example, the comparison is a trivial example where the behavior needs only a single operation. The second example is non-trivial, and the comparison may need the check the full robot path against all defined walls.

Each dynamic constraint falls under two main categories: explicit or implicit. Implicit requirements are those assumed to be held by the design domain. These are defined by the DSML designer, and are non-modeler configurable. Such constraints are assumed to always be held under any model in the domain, regardless of the modeler's design choices. Implicit constraints are therefore domain-specific. Explicit constraints, on the other hand, are defined by the modeler, which is often the case when engineers are tasked to meet particular requirements defined by the customer. In such a constraint, the modeler may have particular needs from using the language, thus requiring dynamic constraint flexibility. Explicit constraints are therefore model specific. An implicit constraint may be read as "All models should behave like X", whereas explicit constraints may be read as "This particular model should behave like X". Take for example traditional LTI controller design, where it is expected that all controllers have an intention to produce stable dynamics, but different systems may have different settling time requirements. The distinction between these categories is important to consider when designing a language to fit the verification feedback framework since explicit constraints must be presented to the modeler, i.e. modeled in the metamodel. On the other hand, implicit constraints may be defined either as a part of the automatic correction framework or by the chosen set of verification tools.

3.1.5 Expert Block

The expert block is responsible for deciding the correct transformation to evolve the model into a more correct model. This step occurs if the model has constraint violations, i.e. e is not an empty set. If e is not empty, then the expert block

represents a domain mapping from the set of constraint violations e to a change in the model u. Expert blocks decide on the best horizontal transformation that focuses on either a change to the model's structure or a change in particular model parameters. Each expert block represents the codified knowledge of a domain expert, who understands how to modify the model based on a verification failure. Such knowledge may be as simple as tuning a value, or more advanced by deleting or adding components. Not drawn in Figure 3.1 is a mapping from the model to the expert block. In some cases, the verification output may be too trivial to make an intelligent decision, and thus model context may be necessary to determine the correct model transformation. Also as similar to the verification tool block, the expert block may represent a composite set of multiple experts. Important to note is that expert blocks will vary greatly based on the domain. Designing general rules of thumb for a breadth of domains is a complex task due to the complicated mappings and structures between the model, requirements, and verification tools. Thus generally the expert blocks are needed to be highly custom algorithms for each DSML.

There are two main categories of expert blocks, either assuming a reactive algorithm or a memory-based algorithm. A reactive type of algorithm makes a change simply based on the current constraint violation. Generally these may be used for simple corrections, or simple mappings between model parameters or structures to constraint violations. In this case, any past model modifications are not needed for future model modifications. Reactive techniques may not be suitable in all cases, and memories of past changes may be needed. In such a case, keeping a track of prior modifications may need to be stored for future modifications. A classic example in control theory for a memory-based algorithm is the Ziegler Nichols tuning method. In this algorithm, a controller gain is adjusted until a particular result from verification is achieved, before tuning a separate gain until a different quality from the same verification is achieved. This is effectively a multi-mode setup, where each

time the expert block executes, the prior state of modification needs to be known.

Three further classifications of expert blocks involve the method of evolving the model towards a verified solution. Such expert blocks may be analytically driven, heuristic driven, or data driven.

1. Analytic methods exist in many design domains, where models have direct mathematical derivations for design solutions. In this case, the model most likely needs to be provided to the expert block for analysis. There can be some simple cases where the constraint violation directly translates into a model evolution solution using a mathematical formula. Analytic design techniques can provide optimal solutions with one model evolution step. Though analytic modifications may have the power to provide optimal solutions, many domains are unable to provide full analytic environments.

2. In systems where no analytic solutions exist, often engineering methods involve some sort of best practice or rule of thumb. This is often the case where the mathematical models involve non-linearities, and no closed form solution exists. This is where the importance of verification tools fit, to test models that cannot be solved analytically. Rules of thumb are then determined through trial and error to produce a set of design heuristics. Not all design domains may have such established heuristics. Worse yet, complex multi-domain environments like CPSs may involve unique domains, and thus have no rules of thumb for design.

3. If analytic and heuristic-based methods fail to define an expert block, the next step is to look into data-driven solutions. Such solutions may be based on trial and error, and could potentially involve some form of machine learning-based solution. Such machine learning could be automated through model mutation akin to a genetic algorithm, and the process could be supervised by the same verification tools for DCF. Similarly, data may be collected on what domain

experts perform to modify models after presented with constraint violations, and their typical decisions may be codified.

3.1.6 Model Transformation for Correction

The model transformations, as connected in the DCF loop after the expert block, are intended for model modification. In Figure 3.1, representing a control system, the model transformation maps the control input u provided from the expert block, and provides the process of model modification. Design of this set of model transformation possibilities is based on the metamodel of model x, with target models fitting under the same metamodel. Thus the model transformation needs to be aware of the metamodel construction. As before, the model transformation block represents a set of possible transformations that may be directly tied to particular expert blocks. The transformations are a simple set of modification rules that dictate the possible model changes Δx that can be applied to the model x.

When adapting a prior language to employ DCF, model modifications need to be specified. Generally, a DSME provides the framework for model transformation, but further development is needed to design model transformations into a prior metamodel.

3.2 Considerations of the Framework

The described components in the framework from Section 3.1 provide the set of classifications and features of each component required for DCF. Looking at the framework as a whole, various properties and development techniques can be established. The framework aims to be specific for DCF based modeling languages, but generic to accommodate a wide variety of domains, verification tools, and model correction techniques.

3.3 DCF Based Modeling Language Design Patterns

Similar to software design, certain patterns for metamodels could be established depending on the domain, DSME, and set of verification tools. Such patterns will aid in reuse of solutions to well-established domains. There exist many correct methods of metamodel design for a particular domain, but generally a particular practice may be more adopted than others. The same is for general software design. In a software project, unrestricted implementation may fulfill the requirements, however not making use of well-known patterns adds to design and development time. Reusing well-established design methodologies in a particular software domain greatly aids in the success and reduces the cost of development. For DCF, such patterns may be established for metamodels, transformations, and verification tools to aid the adoptions and success of DCF based modeling languages. Such patterns are an effective constraint on the language syntax where many valid syntax definitions exist with proper semantic mappings. Using patterns lets the designer focus on the semantics rather than both the syntax and semantics.

3.3.1 Metamodel and Transformations

There has already been an effort in establishing sets of metamodel design patterns [24] [2]. The same is true for model transformation patterns [64] [17] [47] [36]. In the context of DCF, design patterns in metamodels and transformation can be targeted towards known domains to better guarantee DCF success. Some languages in different domains may share similar metamodels even though the domains seem quite different, such as defining finite state machines or hybrid controls. An accepting end state may be similar to an acceptable final mode. Thus defining how an accepting state should appear in the metamodel for the domain of finite state machines may translate similarly into the domain hybrid control design. Similarly, the set of model transformations to correct the model for a hybrid controller may

also translate to the corrections needed for a finite state machine. Having a set of metamodel patterns with model evolution techniques can then potentially translate amongst multiple domains, and therefore have a broad impact. This can let a DSML designer focus not on the exact abstraction method for a domain, but simply reuse established methods and then shift focus to model correction methods for their design techniques.

3.3.2 Verification Tools

Though individual verification tools in similar domains provide slightly different qualities regarding verification output, different tools in similar classes aim to verify the behavior of identical domains. Thus verification tool input abstraction may be classified under different patterns. Having particular verification patterns also implies the potential for ready working interpreters. This can greatly reduce the adoption time of DCF by reducing the development time of verification interpreters. As taxonomies of verification tools are developed, input design patterns can be provided as part of a tool selection suite for quick integration. Focusing on tool selection also may provide the necessary design constraints for the metamodel, and therefore the following model evolution transformations.

3.4 DCF Based Modeling Language Properties

The high-level DCF framework in Figure 3.1 represents a system construction with a potential set of dynamic properties. Such properties describe the qualities of project development, rather than the qualities of the product. One of the first questions to be asked of the framework is regarding the level of automation. A second question, assuming the DCF is fully automated, regards how the models evolve over time to satisfy the requirements.

3.4.1 Level of Automation

The level of automatic model correction is dependent on the relationship between the dynamic constraints, verification output, and expert blocks. Full automation in this sense does not consider potential limitations of the model correction process. For example, some verification tools may require a manual entering of data into a UI, but it is assumed that if proper verification transformations exist then no design choice is made. Therefore full automation means that a user does not need to make a decision in the correction process for full automation. A DCF DSML can therefore be fully automated if there is a full mapping between verification output and the dynamic constraints, and expert blocks exist to handle all possible dynamic constraint violations. There may very well be implementations where an expert block is unable to be determined for a subset of the dynamic constraint violations, and therefore may only be partially automated. This may be especially true in highly complex design domains. The focus of this framework is to provide the possibility of full automation, agnostic to a particular domain.

3.4.2 Automatic Correction Properties

Due to the similar negative feedback connection as inspired from control systems, particular properties may be notable when designing a DSML with verification feedback correction. This raises questions on the design dynamics of the system, since iterative development through model evolution is similar to a dynamic system state over time. Looking at basic properties of control systems, similar properties of the design process may be speculated. Such speculations are not a perfect mapping between the DCF framework and control design, but they may exhibit similar properties.

- **Initial Conditions** - Some languages may require that an initial model be provided, or that the initial model is already relatively close to a verifiable

solution. In rare cases, it may be possible that no initial model needs to be provided. This is a particularly interesting case since this implies that only the set of requirements needs to be provided to generate a model that passes verification. In most instances, it is expected that some base model is needed since model transformations may only operate on model modification, not necessarily model creation.

- **Set Point** - The set of requirements is similar to the setpoint input to a control system. There may be many instances where the set of requirements is unattainable. In such cases, the requirements may be considered to be invalid. For explicit requirements, there may be many instances where a non-domain expert modeler does not understand that being too strict on requirements has no solution.

- **Convergence** - Depending on the initial model, the set of defined constraints, and the expert block, models may converge to meet requirements. In many cases models may never be able to converge and design changes may need to occur either on the expert block or from the language user by selecting less aggressive constraints. Ideally, convergence for a model implies that all verification passes. There may be cases when the model may never converge to a solution, or simply take too many iterations to solve. A model not converging is parallel to a control system having a non-zero steady-state.

- **Divergence** - As opposed to convergence, a divergent system may occur in some domains when the initial conditions are too far from a design equilibrium, the setpoints are too aggressive, or if the expert block is too aggressive. The expert block being too aggressive is similar to control systems where the gains are set too high, causing oscillations or unstable dynamics. In the context of model evolution, divergent properties may be manifested either as values running away, or additional model nodes being added infinitely. In the oscillatory

divergent case, the first method of correction may drive the model to fail other verification, and a following different expert block may then drive the model back to an even less correct state from before, to be driven by the original correction step.

- **Over Dampening** - An over-dampened control system implies that the controller may be sub-optimal regarding settling time. Similarly, with DCF, an expert block may make changes that only have minimal impact on the model, requiring further model iteration before passing constraints.

- **Under Dampening** - An under-dampened control system implies that the controller may overshoot the goal before settling to a steady-state solution. Similarly, with DCF, an expert block may make changes that only have minimal impact on the model, requiring further model iteration before passing constraints. There may be cases when the model may never converge to a solution, or simply take too many iterations to solve.

CHAPTER 4

The Dynamic Constraint Feedback Metamodeling Language

Chapter 3 provides a high-level functional view of the various required components for a DCF featured DSML. In order to increase the adoption of DCF featured DSML, the framework needs to be concretely and practically defined. With this in mind, here is a short review of what has been covered.

1. Implementing DCF, as presented, involves a structure of connected components, but the exact implementation per domain may vary.

2. Implementing DCF requires knowledge of the DSML's metamodel for model transformation and constraint definitions, i.e. the metamodel is in a similar abstraction layer.

3. Dynamic constraints need to be modeled in conjunction with model transformations and verification tools.

4. A DSML aids design processes through focus on modeling domain concepts rather than low-level details.

5. A DSME imposes structural constraints by only allowing model creation based on the metamodel.

Design of a DSML is in itself, a design domain: the domain of modeling language design. Similar to how a DSML provides the benefits of structural constraints on a model, a DSME has rules for metamodel design. Implementation of DCF in a DSML is possible but has the potential for an unrestricted implementation. Design of a DSML with DCF therefore may have a higher success rate by enforcing DCF-based

structural constraints during DSML design. Thus, the philosophy of improving modeling design through the use of a DSML may be applied to the design process of a DSML itself. In other words, the principles and benefits provided by a DSML during model design may also apply to the design of DSMLs. This is effectively what DSMEs provide: a pre-defined meta-metamodel for the design of metamodels. Structurally, metamodels are constrained by the meta-metamodel, thus enforcing a certain structural quality to metamodel design. A DSML may therefore be created for the design of DCF based DSMLs, by rethinking the meta-metamodel to account for DCF qualities. DSMEs also provide interpreters to produce the set of model builders and template interpreters for a DSML, and such tools may also be useful for a DCF based DSML. This new DSML language is called the Dynamic Constraint Feedback Metamodeling Language (DCFML).

When considering implementation of DCF in a DSML, particular aspects in Chapter 3 need to be implemented. These include the modeling methods necessary for capturing constraints, mapping verification tools to constraints, and defining model transformations for model correction. If carefully thought out, these may be implemented in any standard DSME. Current DSMEs such as WebGME, as taken from the name, are intended to be generic to model anything, including a DCF featured DSML. However, as similar to developing low-level artifacts by hand, it is easy to fall into unrestricted development, potentially making DCF a messy integration. Certainly, this does not mean the resulting DCF integration methods will not be effective, but rather that they can be difficult to implement. Also with the prior abstractions described in Chapter 3, the wheel should not need to be reinvented for each DSML to implement DCF.

Therefore it follows that implementation of DCF during DSML development can benefit from the use of both interpreters and structural constraints. The DCFML, presented in this work, replaces the standard WebGME meta-metamodel. The structural constraints from the DCFML can ensure that constraints are modeled and con-

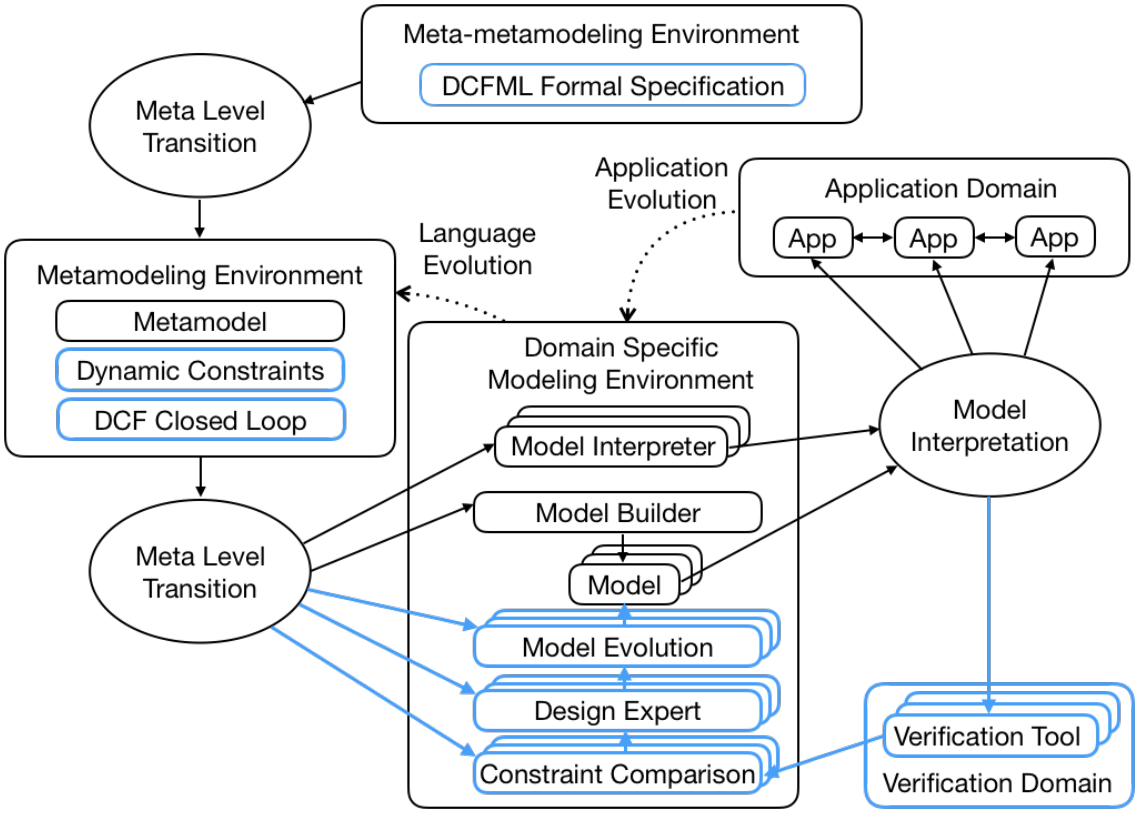

Figure 4.1: The updated MIC diagram from Figure 2.1, showing integration of DCF with the DCFML.

nected to the appropriate verification tools and design techniques. Similarly, model correction transformations may be designed with knowledge of the metamodel and may be tied to design expert blocks. An interpreter can then ensure that a produced DSML follows a well-established structure.

Figure 4.1 shows an updated version of the DSML design diagram from Figure 2.1, highlighting the new components from the integration of the DCFML. The blue highlighted components represent the new components. This abstraction demonstrates the DCF closed loop by using new model interpreters to produce inputs for verification tools. The verification tools then provide the information for constraint comparison, which sends constraint violations to the design experts. The design experts then provide the model modification necessary to evolve the model,

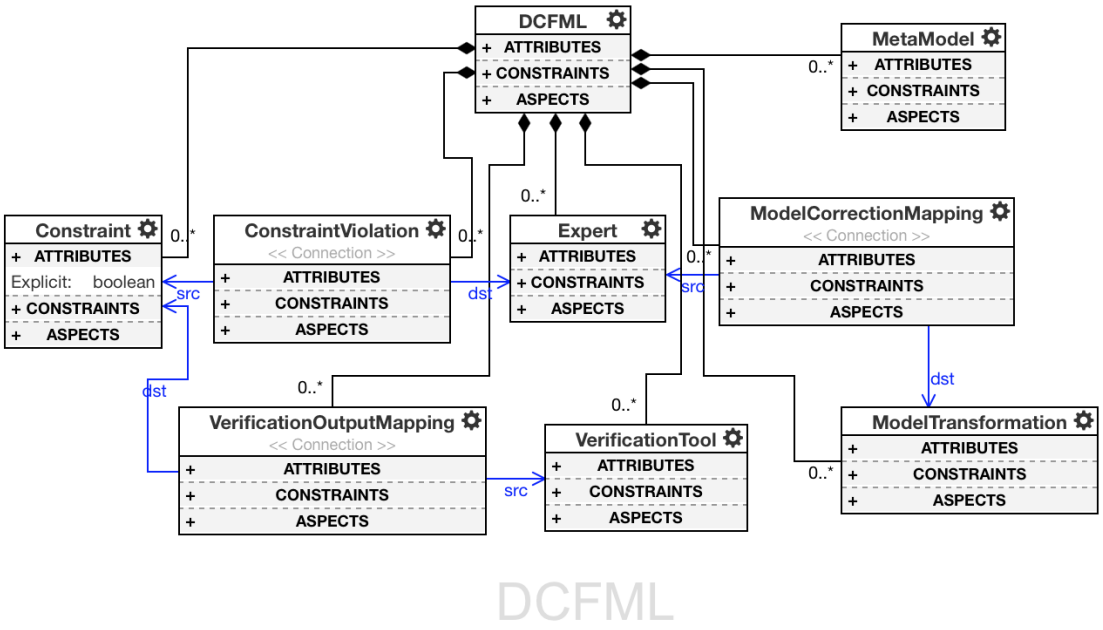

Figure 4.2: The meta-metamodel of a modeling language considering dynamic constraint feedback.

from which the process may repeat. These components, other than the verification tools, are produced from the Meta-Level Transition from the metamodeling environment, which retains the original metamodeling design method. In addition, dynamic constraints and the rest of the connections needed for closing the DCF loop are defined in the similar abstraction layer as the metamodel. Such structure is defined in a higher abstraction, at the meta-metamodel layer, and is where the DCFML is defined. A concrete implementation of the DCFML is discussed in the following sections.

4.1 Overview

The DCFML has components that may be thought of as functional mapping between different parts in a component based design. An overview of the framework may be built as a metamodel using WebGME following from the high-level perspective in Figure 3.1. The metamodel for the DCFML is shown in figure 4.2. A main

difference between the two figures is how edges are defined, where the metamodel needs to represent the connections as models representing a mapping. For example, the output of verification tools needs to be associated with corresponding dynamic constraints, so the VerificationOutputMapping model provides this functionality. Some other notable differences are the exclusion of the deployment interpreter and the comparator block. The comparator block is purely implied since it is not needed to create the block for each constraint. Instead, each constraint is mapped to an expert block, implying that a violation of the particular constraint will be provided to the expert. The lack of the deployment interpreter is due to not being a necessary part of the closed feedback loop.

4.2 Metamodel

The metamodel is the formal specification of how models are constructed and drives how model transformations are to be defined. Figure 4.3 shows the formal specification of a metamodel, i.e. the meta-metamodel. This meta-metamodel is designed to replicate how common DSMLs are constructed in WebGME. This meta-metamodel does not capture the full capabilities of metamodels, but is sufficient for the DCF framework in simple languages. First Class Objects (FCOs) are replicated with a similar name, the MetaModel FCO (MMFCO). This is also true for three class relationships for Inheritance, Containment, and Pointers. WebGME provides the ability to decorate objects in a model and even includes their own metamodel style decorator to make models produced under this meta-metamodel appear like a WebGME metamodel.

4.3 Constraints

In the prior works that implemented a method of DCF, constraints were defined either implicitly by the design domain (i.e. controller stability, no deadlock) or they

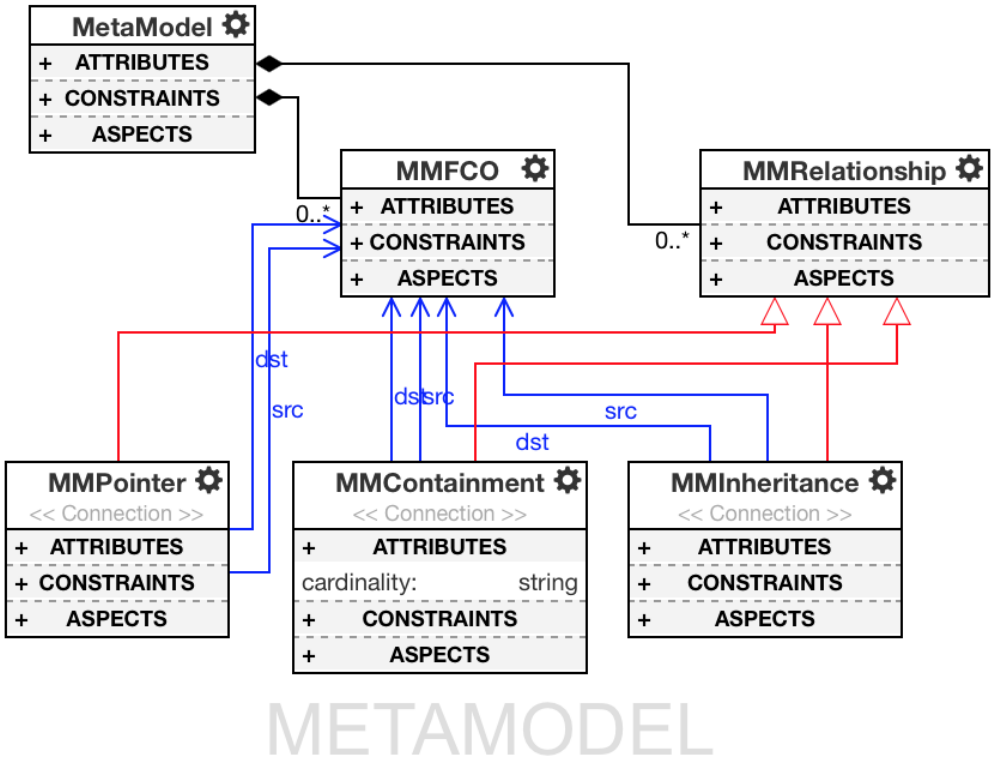

Figure 4.3: The formal specification of a metamodel, i.e. the meta-metamodel.

were defined explicitly (i.e. controller rise time, accepting states). Implicitly defined constraints do not need to be captured by the metamodel since they are assumed to be held for any model, but explicitly defined constraints vary between models and need to be defined. For the DCFML, all constraints need to be defined for closing the loop. Figure 4.2 shows the method of constraint capture, with an attribute to define whether the constraint is explicit or implicit. The constraints also contain a set of datatypes to define the structure of the constraint. Upon interpreting a model of a DSML from the DCFML, explicit constraints become an addition to the metamodel which will be shown in the case studies in Chapters 5 and 6.

4.4 Expert Block

Each expert block in the feedback loop may require a complex set of rules and heuristics. Currently, this expert block serves as a placeholder since modeling an entire set of possible algorithms into a single metamodel is outside of the scope of this work. The case studies will only provide a simple reactive set of expert blocks that will encompass a simple mapping between violations and model transformations. More complex expert blocks may involve the coordination of multiple methodologies and may need to include functionality like history tracking and stopping criteria. An example of this is the Ziegler-Nichols controller tuning method. Here a model needs to be dynamically tested until a particular state of tuning is reached before proceeding with the next phase of tuning involving different gains to be adjusted.

4.5 Model Transformations

For the DCFML, model transformations serve as a method to evolve a model into a more correct solution. An already available transformation language could also potentially be used in place of the transformation definitions defined in this DCFML. The model transformation language GReAT would be a good fit, but support is only

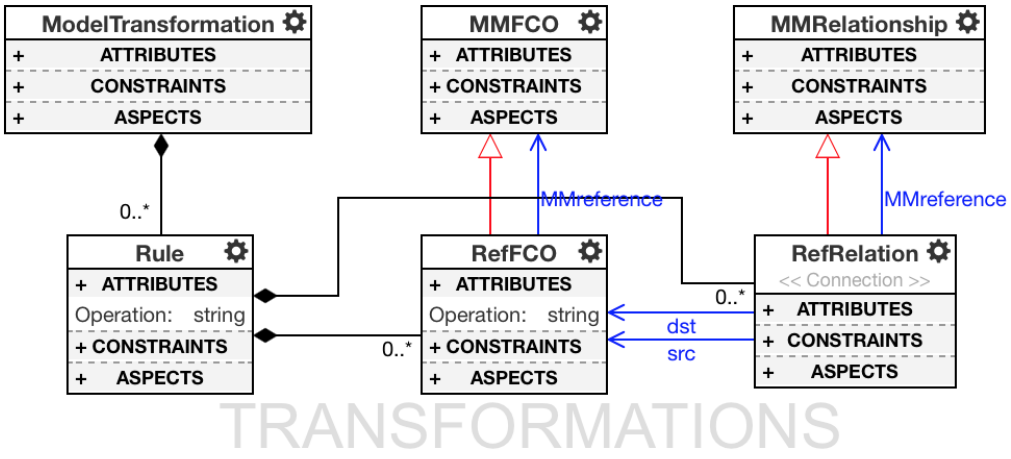

Figure 4.4: Model transformation definitions, relating to components in the metamodel.

available for GME instead of WebGME. The only caveat to using a general purpose transformation language is that they provide methods for many transformations, for any horizontal, vertical, exogenous, and endogenous situation. Since only model evolution is needed for the DCFML, transformations needed are fairly simple due to both the source and target models being under the same metamodel. This is further simplified by limiting the types of transformations to either adding, deleting, or modifying attributes of model components. There are perhaps more evolution techniques needed when considering general domains, but these simple rules are sufficient for the case studies.

Figure 4.4 shows the metamodel for transformations. This metamodel is inspired by simple examples in GReAT. Each transformation method has a set of rules, where rules contain references to the metamodel. Each reference to a metamodel component also has an attribute that defines the function of the reference. These functions include either adding, removing, changing an attribute, or no action.

Interpretation of the model transformation definitions provides source code in the form of functions with inputs relating the references. These functions require the WebGME identifier of the blocks, a necessary part to ensure that the correct

blocks are added and related to existing instances in the model. Each expert block is responsible for deciding which transform needs to be performed.

CHAPTER 5

Case Study: Pathing Language

The Pathing Language is a simple DSML intended for fourth-year elementary school students to design behaviors for a self-driving vehicle. The simplicity of the language serves an example design domain that demonstrates fitting a language into the verification feedback framework. The language's intent was to be deployed at multiple elementary schools among many students, all tasked with designing a trajectory for the CAT Vehicle that would not violate important safety constraints. This served as an educational tool for the children, to learn about issues of non-holonomic motion planning to improve spatial reasoning. To ensure safety without needing any double-checking by a third party, verification tools were implemented to ensure that behaviors operated within safety-critical boundaries. Models of trajectories were then interpreted into code and demonstrated at each school's open fields. Also, models could be interpreted into Lego EV3 code, which corresponds to a scaled-down version of the field for students to run their code on a physical platform inside the classroom.

This language is highly domain-specific, as it is intended for generating code for a particular vehicle to be operated in a particular environment. This results in a modeling language that does not need to capture all aspects of the working system. For example, since the vehicle is known and does not change, this does not need to be captured in the metamodel. Similarly, aspects of the environment also do not need to be modeled. The resulting metamodel and constraints are therefore greatly reduced in complexity. Important safety constraints do however need to be considered since many potential paths may drive the vehicle into obstacles or outside of the constrained environment.

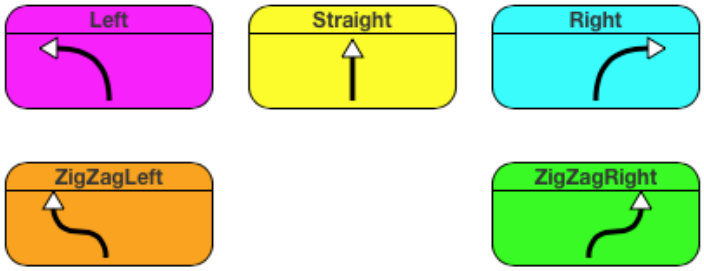

Figure 5.1: The possible set of primitive motions in the pathing language.

This language was developed and deployed before the consideration of the DCFML. The description of the language will be initially presented as it was initially developed before demonstrating how it can be fit into the DCFML. This serves as a case study for adapting a prior language to make use of DCF. The language's simplicity serves well as a preliminary demonstration for the DCFML.

5.1 Domain Definitions

This particular domain is highly specific, considering that only limited choices were allowed for path creation. A path in this regard is a combination of a set of ordered primitive motions. After one primitive motion is executed, a following primitive motion is executed. A complete path may then be thought of as a sequence model, with no branching nor decision making. Figure 5.1 shows the set of primitive motions designed for this language. Each primitive motion has no adjustable parameters, so anytime a primitive motion is invoked the vehicle will drive in the same manner.

Deployment in this domain involves generating necessary code to carry out the sequence of ordered primitive motions on the self-driving car. For the final demonstration, the vehicle would be placed on a large open field before running the path. The primitive motions were designed based on the capabilities of the vehicle, from pre-developed sets of controllers. The chosen motions result in a grid of possible destination coordinates due to the 90 degree turns and driving distances. Figure 5.2

shows a sketch of an open field at an elementary school with the largest grid layout available, a 4x4 square. As an added challenge for path creation an obstacle may be placed at one of the grid points, and the students must create a path that avoids the obstacle.

The metamodel for the pathing language is shown in Figure 5.3. A *Path* is defined as being the combination of *PrimitiveMotions*, and *Prim_Mot_Connections* to describe the ordering. Each *PrimitiveMotion* can be of one concrete type, either *ZigZagLeft*, *Left*, *Straight*, *Right*, or *ZigZagRight*. The metamodel has some attributes like *Distance* and *Velocity* listed, however these were for initial development and served no purpose when deployed. Inside of *Path*, multiple attributes let a modeler define the set of explicit constraint values. Figure 5.4 shows an example model from the metamodel, with a decorator implemented to reduce ambiguity during model design.

5.1.1 Constraints and Verification

Even with a simple language, it is easily possible to design a path that places the vehicle in a dangerous situation. Constraints therefore need to be implemented and validated to ensure no unsafe driving behavior becomes generated code. As shown in the metamodel, some explicit constraints were defined as attributes in the *Path* node. Constraints for this language were defined both explicitly and implicitly, and therefore not all constraints are shown in the metamodel. Since explicit constraints place the responsibility on the modeler, they need to be a part of the metamodel. Implicit constraints still need to be understood by the modeler, but that are non-modifiable.

The first main constraint involving safety is that no path should ever drive beyond the defined grid from Figure 5.2. This constraint is safety-critical, since it is a trivial task to model a path that extends beyond the grid, i.e. four *Straight* motions in a row. Since this applies to all paths, this is an implicit constraint. This constraint

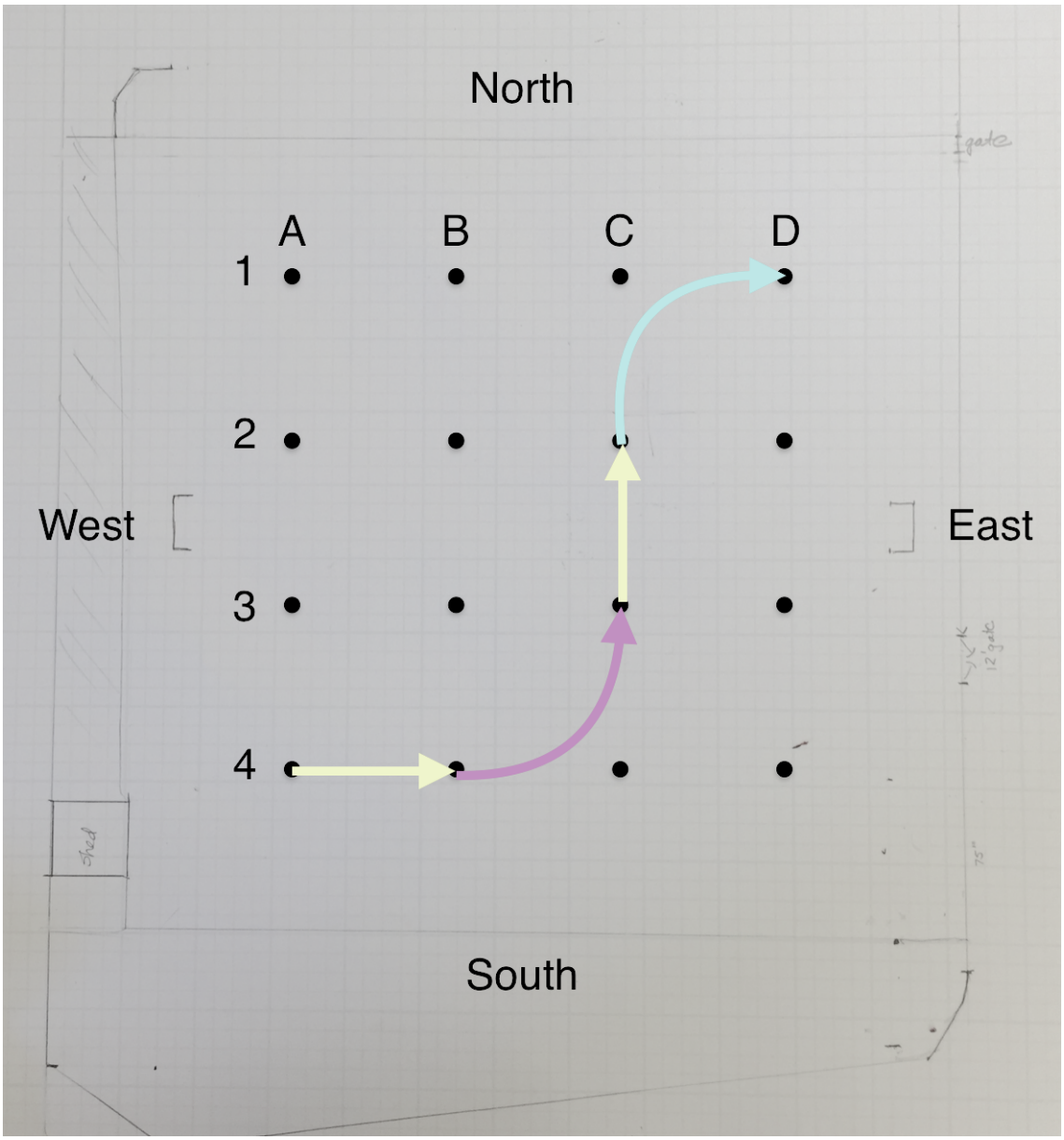

Figure 5.2: An example path using primitive motions within a grid constraint. Shown are the coordinates, orientation, and a diagram of a physical space for running paths.

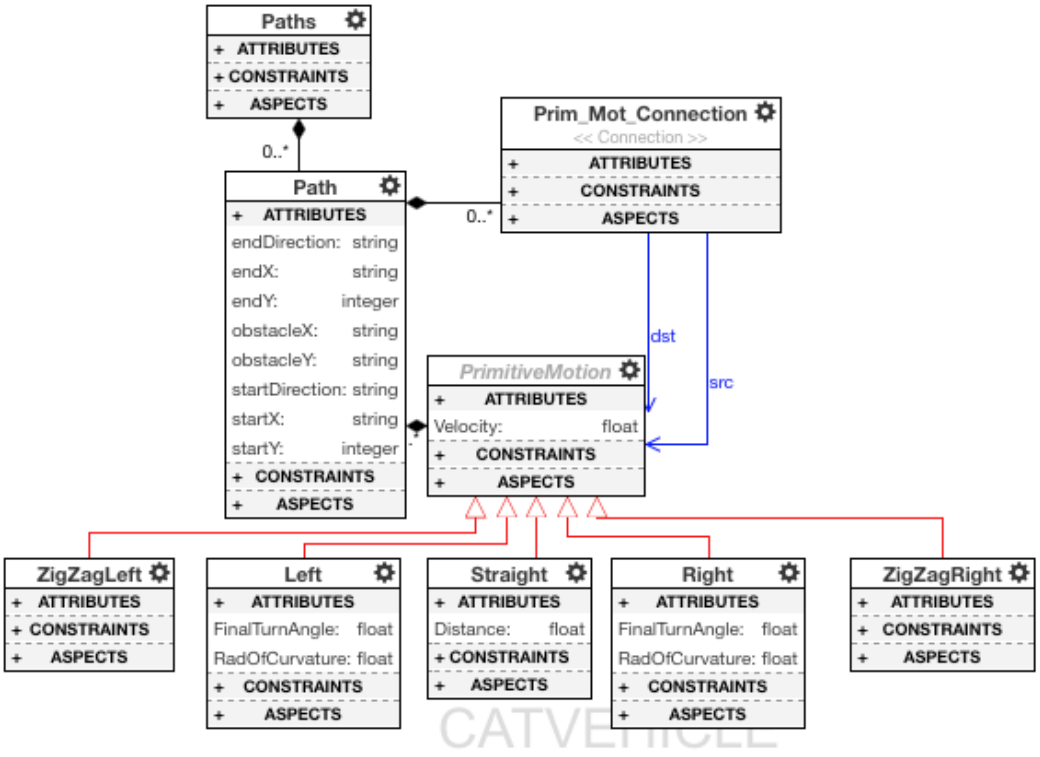

Figure 5.3: The metamodel for the pathing language.

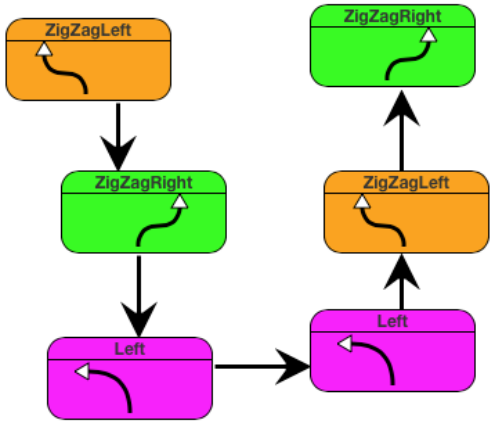

Figure 5.4: An example path created using the metamodel from figure 5.3 in WebGME.

is verified using path integration techniques, which is similar to reachability analysis. Pseudocode for the path integration method is shown in Algorithm 1. If the path ever reaches a grid point outside of the domain's area, then this dynamic constraint is violated.

Algorithm 1 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
```

The next safety-critical constraint involves the obstacle. It was a task for the teacher to place an obstacle on a particular grid point, which the student's then needed to define. Because of this flexibility, this constraint was an explicit part of the metamodel. As similar to the out of bounds constraint, this constraint can be verified through path integration or reachability methods.

The third constraint is viewable in the metamodel, and is therefore explicit. To ensure that students had a well-formed path with a properly defined starting point, the students needed to define both the starting and ending coordinates and orientation. For fewer complications on demonstration day, the starting points had to be on the grid corners. For more interesting paths, the endpoint had to also be on a corner. The particular corners were not important, but the verification ensured

Constraint	**Classification**	**Purpose**	**Type**	**Verification**
ObstacleAvoid	Explicit	Safety	Coordinates	Path Integration
OutOfBounds	Implicit	Safety	Area	Path Integration
WellDefinedPath	Explicit	Both	Coordinates	Path Integration
MinMoves	Implicit	Education	Numerical	Comparison

Table 5.1: The list of pathing language constraints, classification in the DCFML, purpose, data types, and verification methods.

that the path with expected start and endpoints were fully well-formed, serving purposes both for safety and education. Verification is also handled through path integration, ensuring that the end can be reached from the initial starting location.

The final constraint is intended to avoid paths that are too simple and is therefore a purely educational constraint. To avoid a straight path from one corner to another (i.e. three *Straight* motions), a minimum number of moves was required for each path. This lets students flex their creativity but serves no safety purpose. No path should be under six total moves, therefore this was an implicit constraint. This is the simplest constraint to check, as only the number of primitive motions needs to be compared.

Table 5.1 shows all of the constraints for the pathing language, along with the corresponding classification, purpose, type, and verification method.

5.1.2 Deployment

Deployment interpretation occurs when invoking the interpreter plugin in WebGME, however code would only be generated after verification was passed. This verification step is checked on each attempt to generate code to ensure that only verified models would produce code. Feedback was provided to the modeler upon plugin invocation, either being a green box for a pass or a red box for a failure. If the model passed verification, code for both the self-driving car and the Lego EV3 robots could be downloaded. If verification failed then no code was available for download, but clicking on the error provided the exact verification failure. Figure 5.5 shows three

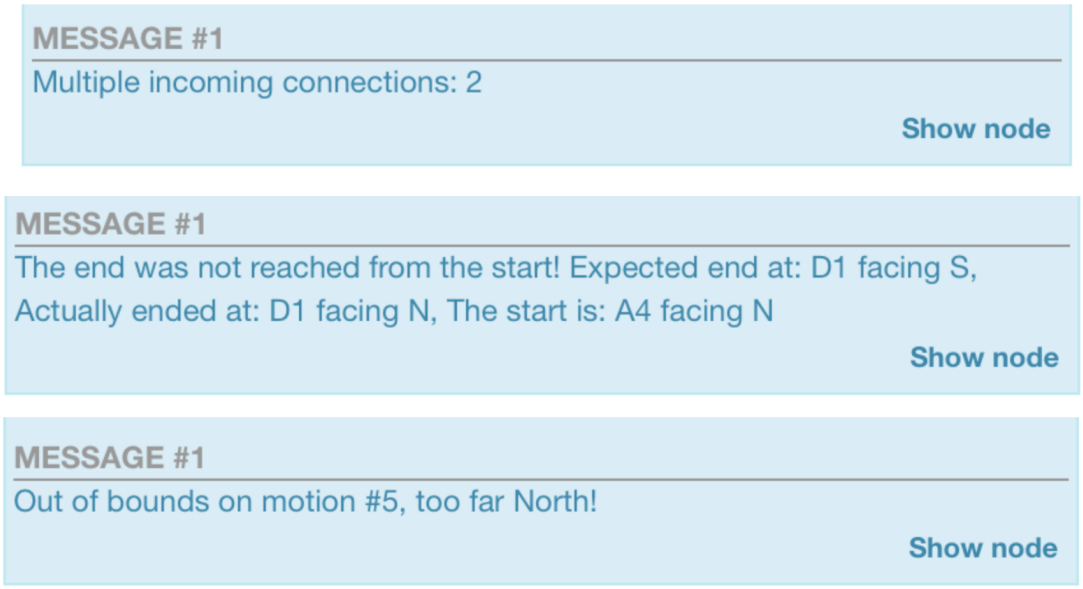

Figure 5.5: Example feedback provided to the user after verification failure.

examples of different verification failures, providing guidance for the students to correct their models. Part of this feedback also provides information regarding some structural errors if pathing ambiguities exist.

5.2 Transitioning to the DCFML

This language originally had most of the pieces needed for DCF, however does not have a fully closed loop. The pieces missing include the model transformations and design technique expert blocks. Fortunately, the language is relatively simple, where the model evolution steps only exist as adding or deleting primitive motions. Also since the language was priorly created in WebGME, transitioning to DCFML is a trivial task since the metamodel is directly compatible. Other parts still need to be designed, such as the model transformation definitions and expert blocks. Also, the constraints now need to be formally defined.

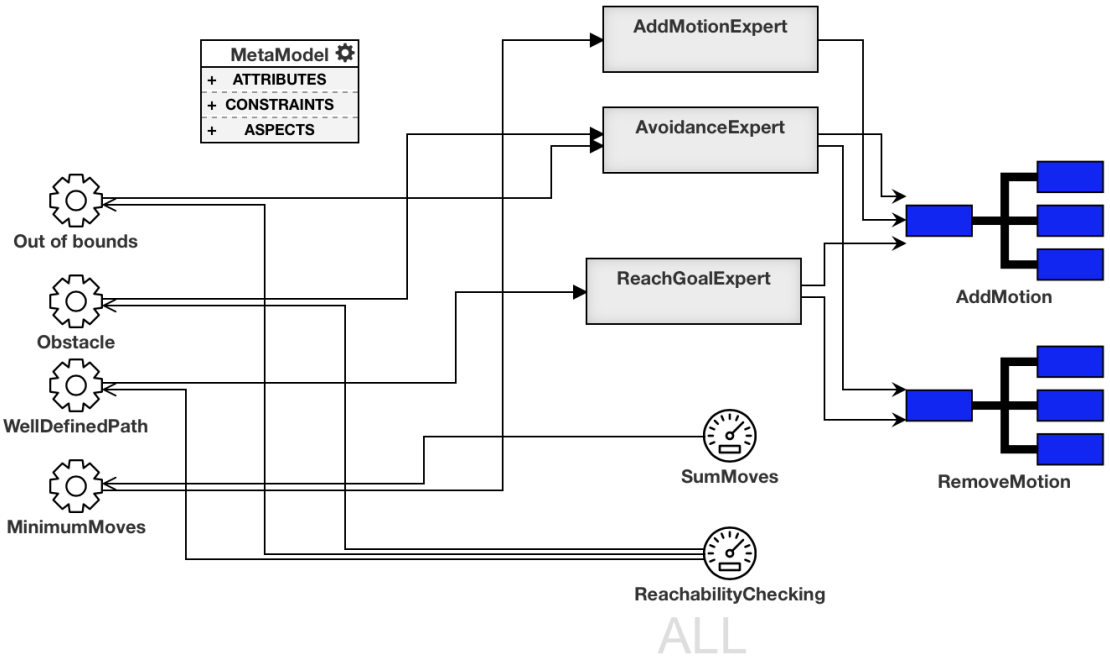

Figure 5.6: The model of DCF using the DCFML for the updated pathing language.

5.2.1 Constraint Remodeling

Constraints exist in both an implicit and explicit aspect, but only the explicit constraints were modeled in the original pathing language metamodel. When transitioning to DCFML, all constraints need to be modeled within the same aspect. Constraints have attributes that describe whether they are implicit or explicit. Figure 5.6 shows how the DCF enabled pathing language appears with the new constraints. A decorator is used in the model builder to help distinguish between different DCF components. The metamodel appears disconnected, however it contains the meta specification needed to define the transformations.

The DCFML provides minimal support for constraint definition, which is sufficient for the pathing language's constraints. Figure 5.7 shows the modeled constraint for the explicit constraint needed to ensure that paths are well defined. This constraint requires the definition of a start and end location and orientation, which is

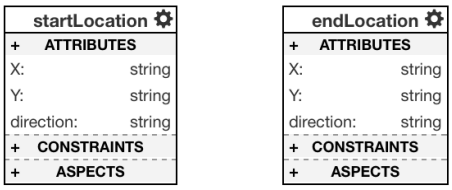

Figure 5.7: The model of the constraint for determining a well defined path.

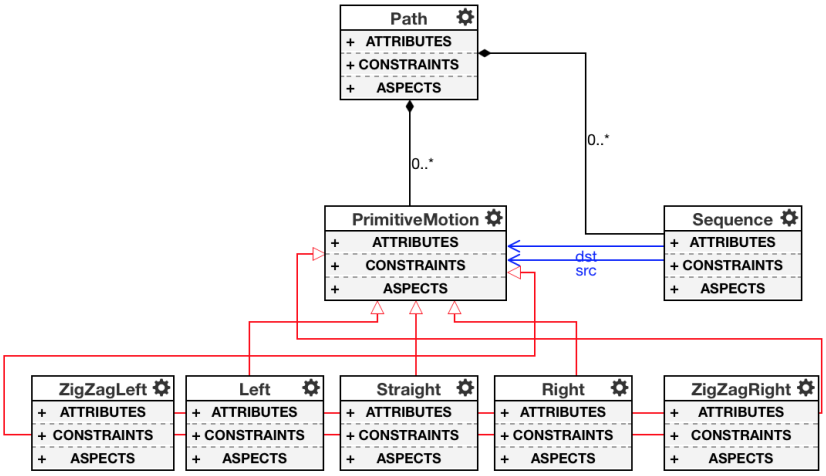

Figure 5.8: The updated pathing metamodel using DCFML, with removed constraints.

seen defined as vectors named startLocation and endLocation.

Furthermore, the original metamodel does not need to have the dynamic constraints modeled since the DCFML specifies them elsewhere. As a result, the Path node can have reduced apparent complexity in the metamodel. Figure 5.8 shows the updated metamodel with removed explicit constraints. Also, the metamodel is slightly changed from removing deprecated parameters, along with a minor refactoring and a renamed connection relation.

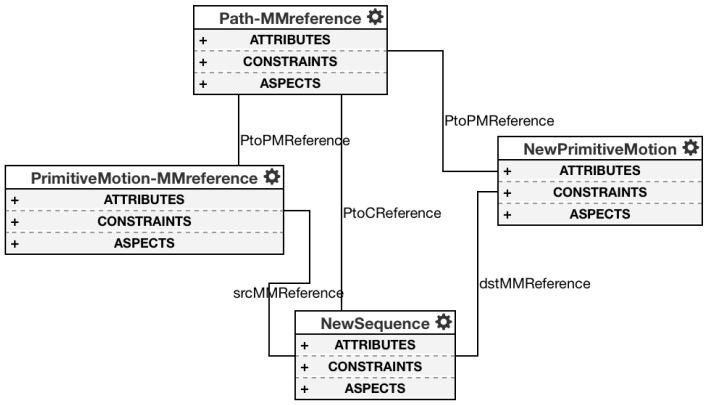

Figure 5.9: The model transformation rule for *AddMotion*.

5.2.2 Model Transformation

As shown in the DCF model in Figure 5.6, only two types of model modifications exist for the pathing language. *AddMotion*, as the name implies, adds a connected motion to the sequence. Conversely, *RemoveMotion* removes a motion and connection from the sequence. Figure 5.9 shows the transformation rule definition. Before adding a new motion and connection, the parent of the newly added models needs to be known, along with a reference to a current motion for a proper connection. All of this is modeled in the transformation rule. *RemoveMotion* is modeled in a similar fashion, however the reference has an attribute that designates the motion and connection for removal.

5.2.3 Expert Blocks

The pathing language is relatively simple, along with the transformation definitions. It would seem that the expert blocks would also be simple. Unfortunately due to the constraints, the expert blocks are not as trivial as simple as the language. For the sake of brevity, only the expert block for the obstacle avoidance constraint violation will be discussed.

Per the language deployment and corresponding developed curriculum, it was deemed that a modeler would design an initial path, then an obstacle would be placed at the second-to-last reached grid point. This provided to be a small design challenge for the students to work around the added constraint. Automating this process requires an algorithmic implementation to account for all of these possibilities. Fortunately, the grid system is very symmetric, therefore only a small subset of obstacle violations needs to be solved for a variety of potential collisions. This results in only needing to look at one corner since a constraint for the endpoint is to end in a corner. Each corner has only two possible ending orientations. These orientations have a diagonal, mirror symmetry. Since all corners are similar under a rotational symmetry, only one corner at one orientation needs to be considered. For the sake of these examples, the North-East corner was chosen, with a final Northward orientation.

Working backward from the end location, there only exist three valid motions to reach this grid point and orientation. These motions are a *Straight*, a *Left*, or a *ZigZagRight* turn. Initially, it appears that only three potential corrections are needed for all possibilities. First look at the case with a final *Straight* motion. Backtracking to the second-to-last grid point, an obstacle prevents the path from reaching this location. The straight path was initially valid, but now that the grid point is invalid the end must be approached from a different grid point. Thus, the final straight motion must be changed to either a left turn or a right zigzag, in order to maintain the goal orientation. Unfortunately, the second-to-last motion now also needs to be changed for path realignment. Figure 5.10 shows the two trivial solutions for path correction based on the last and second-to-last motions in the path. This solution is relatively straight forward in the cases of a second-to-last motion involving either a straight or a zigzag right turn, as both motions can be easily changed to avoid the obstacle. For the expert block, each of these can be corrected by invoking two RemoveMotion transformations, followed by two

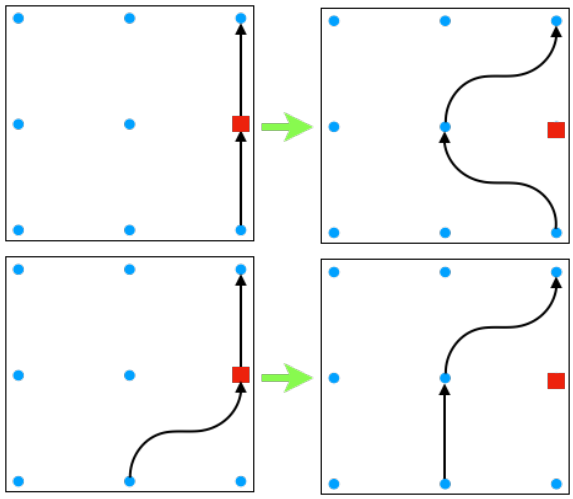

Figure 5.10: Example trivial re-routing solutions for problems with a final straight motion and a priorly placed obstacle.

AddMotion rules.

However, there exists a third option for the second-to-last motion in the case of a final straight motion: a left turn. The orientation and location at the start of the left turn scenario have no available primitive motion to connect to the final motion. As a result, a further step back is needed to check for further routing possibilities. At this stage, all five primitive motions can have a valid lead into a left turn. All five of these scenarios need to be evaluated for routing possibilities since each starts at a unique pose. Figure 5.11 shows all five scenarios along with five corresponding solutions. As opposed to the initial two trivial scenarios, some of the solutions now change the number of motions. Two scenarios remove a motion, and one scenario adds two motions. In total, the final straight scenario resulted in needing seven potential design corrections.

There are now two scenarios remaining. The final left motion leaves the second-to-last grid point non-adjacent to the boundary, therefore having a prior motion existing as any of the five potential motions. In all of these corrections, the only valid final motion that avoids the obstacle is a straight motion. Figure 5.12 demonstrates trivial solutions in the case of a second-to-last straight, zigzag left, or left motion.

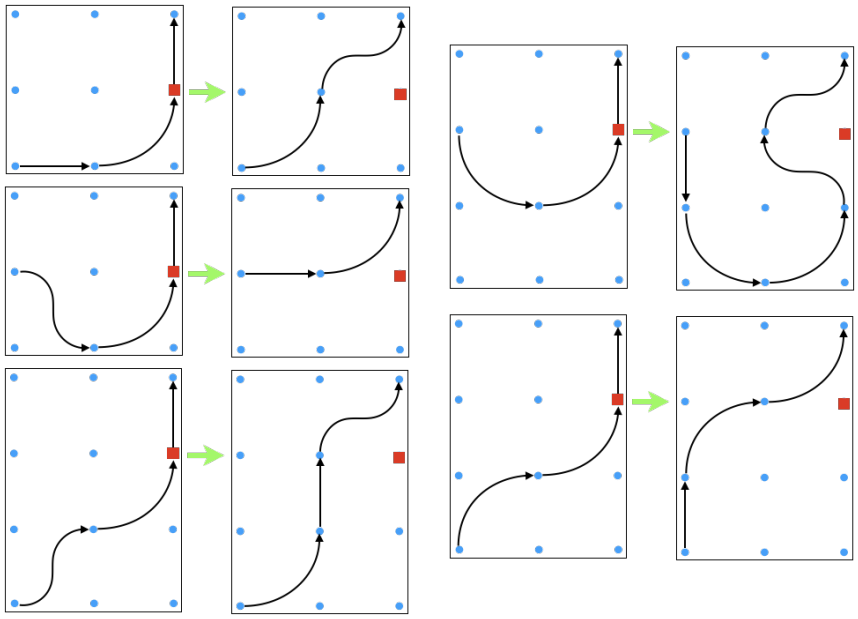

Figure 5.11: Followup non-trivial re-routing solutions for problems with a final straight motion and a priorly placed obstacle.

Though these are trivial, in all cases the total motion count needs to increase by either one or two motions in order to reach final straight motion with the correct orientation. The case of the leading right or leading zigzag right is more cumbersome since the limited set of primitive motions with obstacle location require large changes in the overall trajectory. As shown in Figure 5.13, the leading zigzag right motion path now must loop around the obstacle, causing two path intersections to reach the final straight. This scenario adds an additional eight new motions. The leading right motion needs to follow a similar correction but adds nine total motions to the original path.

The final scenario involves the final zigzag right motion. As similar to the final left motion case, the only valid final motion to avoid the obstacle placement is a straight motion. Also similar are the possibilities of the second-to-last motion, where all five motions are valid. Figure 5.14 demonstrates the three trivial solutions, involving a second-to-last motion of either a zigzag left, straight, or left motion. These path corrections also represent some of the inverse operations from previous

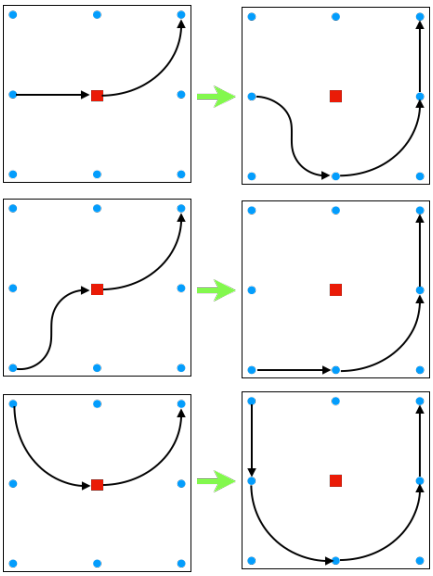

Figure 5.12: Example trivial re-routing solutions for problems with a final left motion and a priorly placed obstacle.

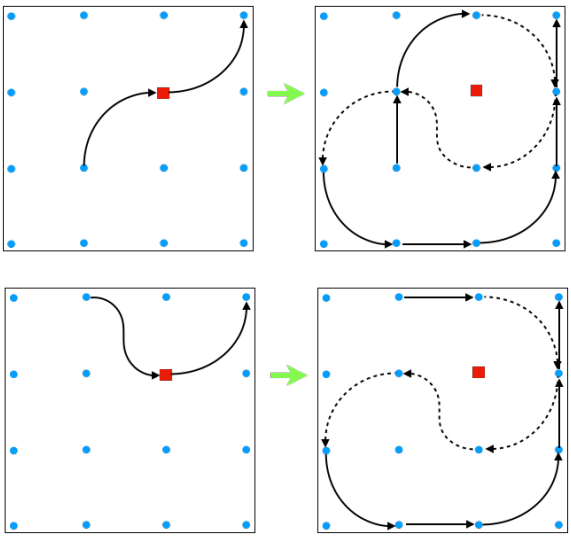

Figure 5.13: Followup non-trivial re-routing solutions for problems with a final left motion and a priorly placed obstacle. Some motions have a dotted line to aid in trajectory clarity.

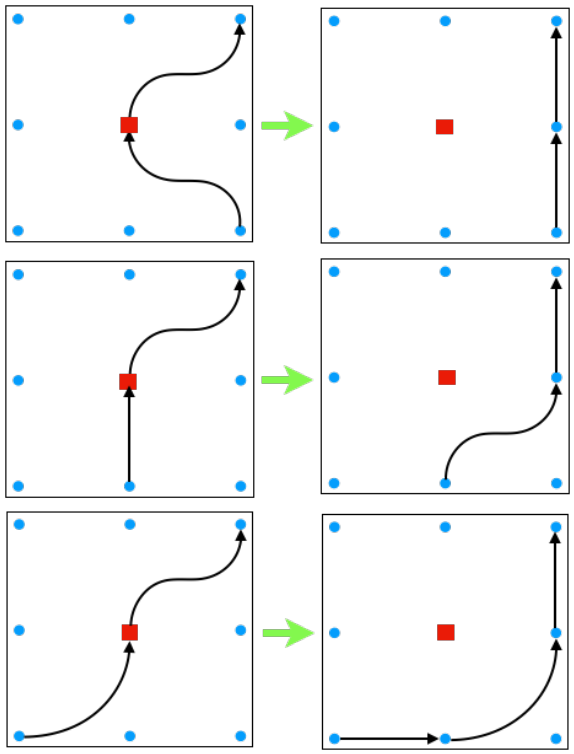

Figure 5.14: Example trivial re-routing solutions for problems with a final zigzag motion and a priorly placed obstacle.

straight corrections in Figure 5.10 and Figure 5.11. The non-trivial scenarios of either a leading right or zigzag right are shown in Figure 5.15, resulting in an additional six or nine moves, respectively.

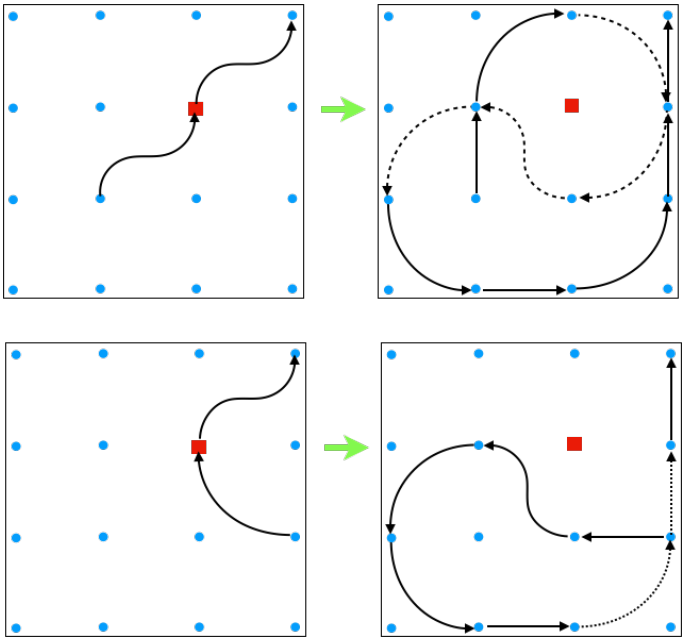

Figure 5.15: Followup non-trivial re-routing solutions for problems with a final zigzag motion and a priorly placed obstacle.

CHAPTER 6

Case Study: Reachability

The reachability language is designed with the idea of using different types of verification tools for a potentially similar task. Each verification tool requires different inputs and produces different outputs, but sometimes these can be incredibly close. Arbitrary, off-the-shelf tool selection will be a key component to the success of the DCFML. This language therefore, as opposed to the pathing language in Chapter 5, is built from scratch to discover the challenges with incorporating different available verification tools. Also, this language's goal is to demonstrate different behaviors of expert block selection, and how they relate to traditional control system characteristics.

6.1 Domain Definitions

This language makes use of hybrid system design through modeling of state transitions and LTI based controllers. Design starts by concisely defining the design domain to ease design complexity. This example DSML is based on controlling an autonomous, Ackerman steering style vehicle using hybrid control techniques to reach a particular goal. Gains for each control mode may be modified, and the time that a controller is active before transitioning into a sequential mode is also configurable. Figure 6.1 shows the model of DCF constructed using the metamodel from the DCFML. This DCF model incorporates two different verification tools, three constraints, two experts, and two transformation techniques.

The control of vehicle trajectory is based on a kinematic model of a car, reduced to the kinematic bicycle model. This method is a common approximation for autonomous vehicles [89] [61]. It is assumed that the car has 0 for all variable initial

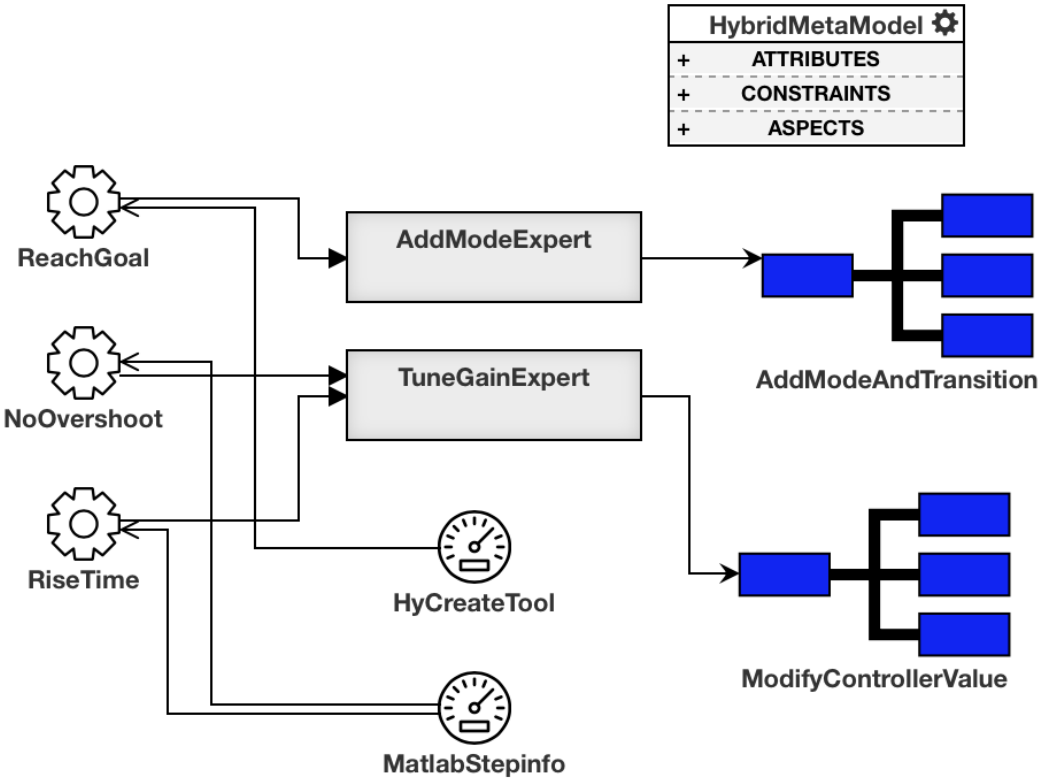

Figure 6.1: The DCF design of a simple hybrid controller language

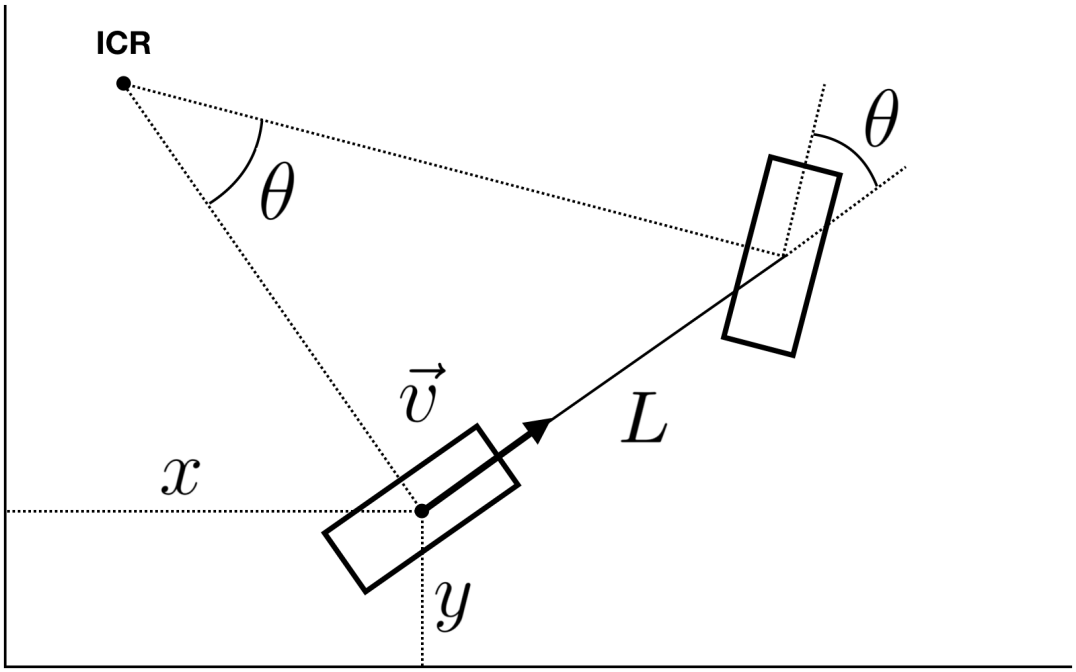

Figure 6.2: The kinematic bicycle model, commonly used for autonomous vehicle approximations [77].

conditions. Only the front wheel is steerable with angle θ, and the angle of the front wheel creates a perpendicular projected intersection with respect to the rear wheel, which is known as the Instantaneous Center of Rotation (ICR). The rear wheel drives operate the speed with control input v. The front and rear wheels are separated by length L. Figure 6.2 shows the kinematic construction. The mathematical model showing the rates of change in position x and y, as well as angular rate ϕ, based on the control inputs are shown in Equation 6.1.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} vcos\phi \\ vsin\phi \\ \frac{v}{L}tan\theta \end{bmatrix} \tag{6.1}$$

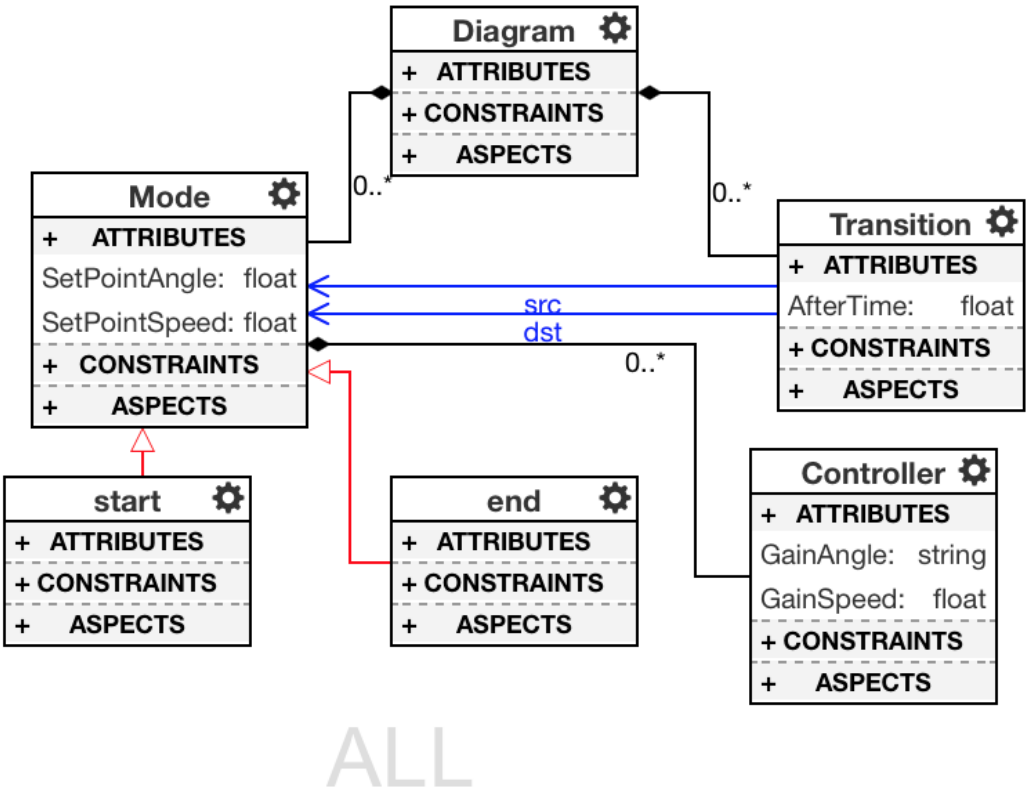

Figure 6.3: The metamodel of a simple hybrid controller.

6.1.1 Metamodel

Inside of the HybridMetaModel block, the metamodel is constructed. Figure 6.3 shows the metamodel definition. This not an all-encompassing method to model all hybrid controllers, but is rather targeted for the simple domain of driving a car to a particular location. An example of a controller a user may construct using this metamodel is in figure 6.4. Each mode transitions to another after a defined amount of time in seconds resulting in modes acting sequentially. Each mode has attributes corresponding to the controller set points for both speed and tire angle. Modes also contain a simple control law, known as a proportional gain, to control steering and speed.

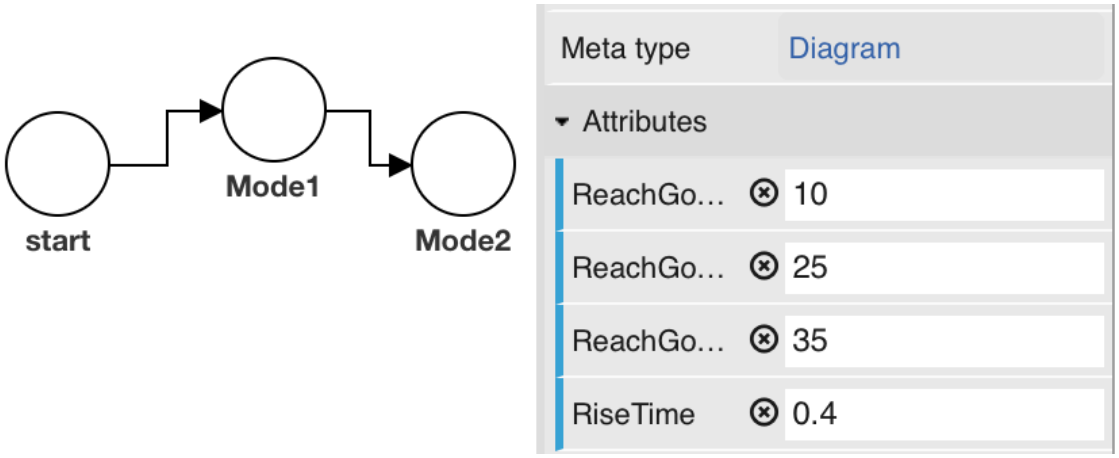

Figure 6.4: An example set of modes in a Diagram constructed using the meta in Figure 6.3. Also shown is the inclusion of explicit constraints that are user definable.

6.1.2 Model Transformations

This example language features two model transformation examples. Figure 6.5 shows a rule inside the AddModeAndTransition transformation. This transformation is responsible for adding a new mode, then connecting a new transition from an existing state to the new state. The new state and transition need to be referenced to both an already existing state, and the parent diagram. Here there are two components that are needed for reference, the original diagram and a state where a new transition is to be connected.

Appendix A shows the code generated from interpretation of the transformation in Figure 6.5. The function shows that the references in the transformation meta result as function inputs, a necessary part to place models under the right parent and to build appropriate associations. Figure 6.6 demonstrates a simple execution of the model transformation on a set of sequential modes. The reference mode is the final mode in the sequence and is highlighted to show WebGME's ID which is used for correct placement when invoking the function shown in Appendix A. After transformation execution a new mode is created along with a new outgoing transition from the referenced mode.

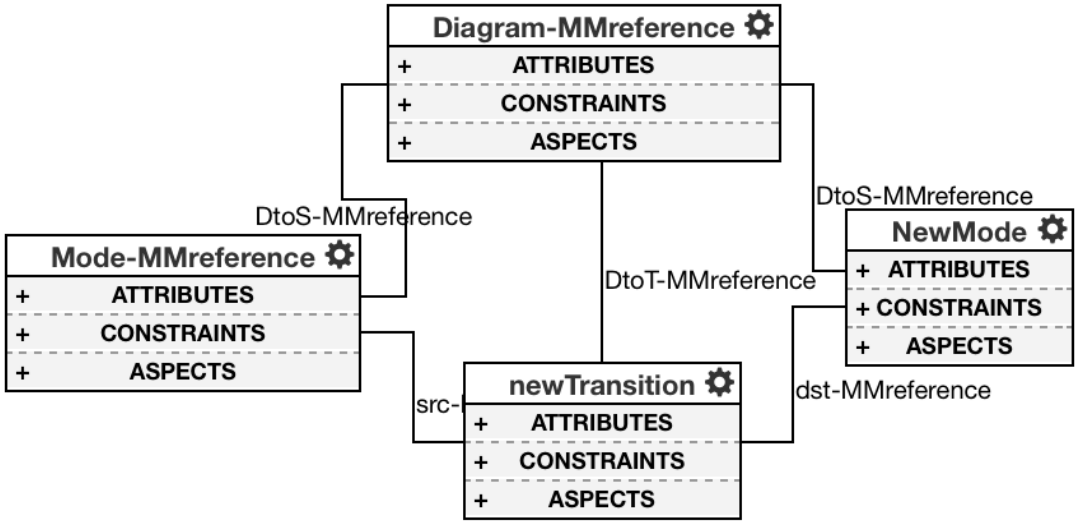

Figure 6.5: A rule in the AddModeAndTransition model transformation using the HybridMetaModel 6.3.

6.1.3 LTI Verification

Constraint modeling is based on the capabilities of the verification methods due to the need for tool outputs to be directly compared. The constraints of the system will be to ensure that a system will be able to reach a particular goal starting from the origin and make sure that there is no overshoot in each controller while being under a particular rise time. Three different verification tools are explored for this language. MATLAB's stepinfo is a simple function to execute and provides simple results that may be directly comparable to primitive constraint data types. For this example language, it is assumed that a 2nd order model is valid to represent the plant of the car as described in Equation 6.2.

$$P = \frac{1}{s^2 + 20s + 40} \tag{6.2}$$

This model of the car is not something that the user can change since this is a language specifically designed for a particular system. Our controller may then have a proportional gain set by the user at an initial value of 100. An interpreter would

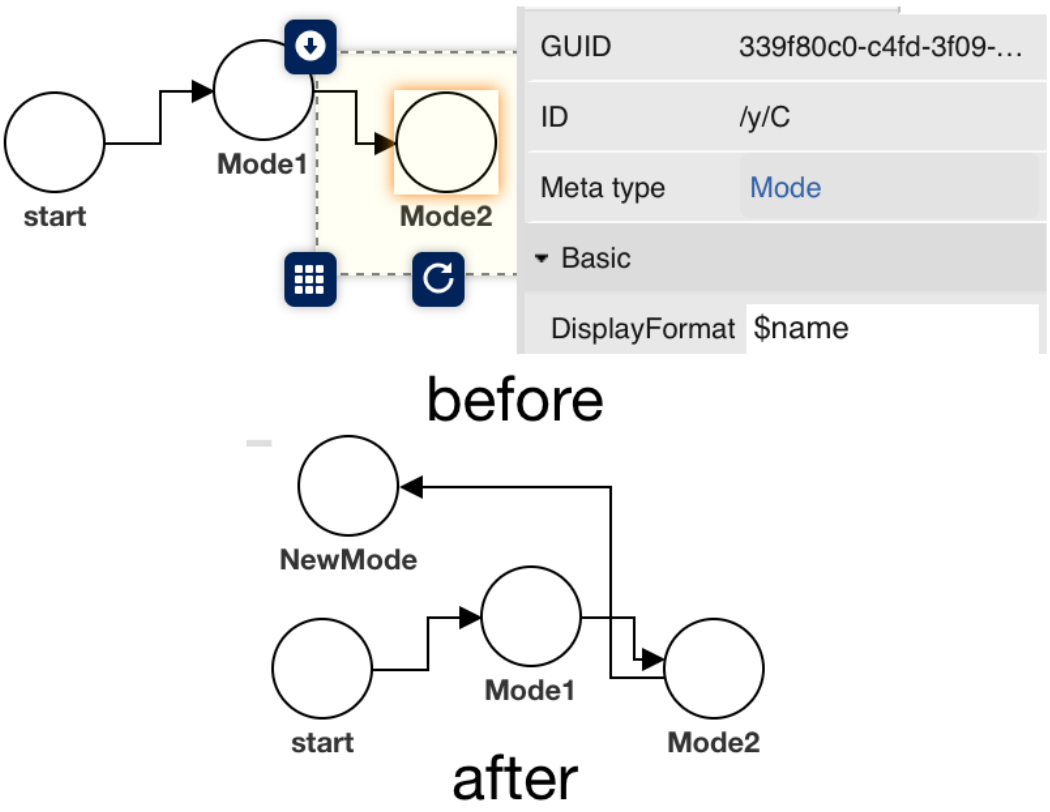

Figure 6.6: An example model before and after the AddModeAndTransform is executed. In the before image, the state of interest is highlighted to show mode of interest, representing the NewMode in figure 6.5.

Stepinfo Label	**Value**
RiseTime	0.2237
SettlingTime	0.3502
SettlingMin	0.6474
SettlingMax	0.7193
Overshoot	0.6962
Undershoot	0
Peak	0.7193
PeakTime	0.4974

Table 6.1: Verification result from stepinfo() defined in Listing 6.1.

then need to be designed to take the prior knowledge of the system plant, along with user-set controller design. The script in Listing 6.1 is an example of generated MATLAB code from the stepinfo() verification interpreter. Executing this script results in the output seen in Table 6.1.

Listing 6.1: Generated verification input for MATLAB's stepinfo().

```
1    Kp = 100;
2    s = tf('s');
3    C = pid(Kp);
4    P = 1/(s^2 + 20*s + 40);
5    T = feedback(C*P,1);
6    output = stepinfo(T);
```

This is a pretty trivial verification tool to implement especially if the plant is not needed to be modeled, however provides directly comparable information against the constraints of rise time and overshoot. One note is that this is only for one particular mode, whereas a hybrid system is usually intended to be built with multiple control modes, thus implying that this tool needs to be run against each mode. This tool is sufficient for checking two of the three constraints, however it cannot handle running the constraint of reachability.

6.1.4 Reachability Verification

Use of reachability tools may appear to be redundant, and obsolete the decision to use Matlab's stepinfo. There are two primary reasons why using stepinfo in conjunction with reachability analysis is still valid.

- Stepinfo provides basic information that directly relates to the constraints and forgoes the need to write data analysis tool to convert data for direct requirement comparison.

- Dynamics of the controllers may be too slow for the transition time, so extrapolation of dynamic behaviors for LTI design has the potential to be incomplete.

Similarly, stepinfo is incapable of providing the result of a hybrid system, hence why the combination of verification tools is important.

There exists a wide selection of reachability tools, however only two different tools were inspected for checking models. For this particular language, dReach and HyCreate were chosen based on their availability and ease of use. dReach and HyCreate provide different methods of computing reachability using a hybrid system. Either of these tools will work for the purpose of this simple language, but there are scenarios where some of the methods of computation and features in each tool may be useful in particular domains. For example, dReach can solve for the specific initial conditions necessary to achieve a strict reachable state.

Both of these tools are based on hybrid control system design, allowing for the modeling of modes and transitions. To generate files for each tool, configuration files can be reverse-engineered for their internal structure. When abstracted, each of these languages model hybrid systems using a different metamodel. Figure 6.7 shows the corresponding class diagrams of each tool. In dReach, outgoing transitions are contained within each mode, whereas transitions are separate entities that manage an association between modes in HyCreate. Note that both of these metamodels also differ from the standard of WebGME metamodel design, where a connective

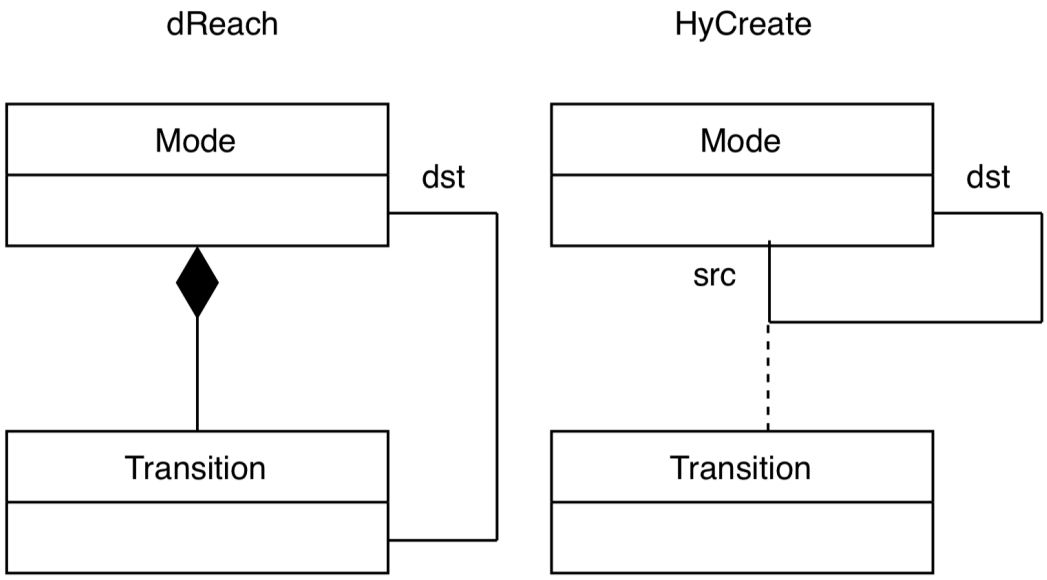

Figure 6.7: Configuration file abstraction for dReach and HyCreate hybrid system models.

model is associated through pointers. All of these different modeling techniques are certainly valid for modeling a hybrid system, but demonstrates the need for model transformation methods to map a model from one metamodel to a different design paradigm.

Each tool also has very different lower-level syntax. HyCreate files are written in XML format while dReach has its own custom yet easily readable syntax. dReach is mostly order-agnostic by referring to different modes through a unique integer. HyCreate requires strict ordering, so each mode needs to have derivatives defined in the same order. HyCreate functions by generating Java code, then compiles and runs the simulation to provide reachability data. Due to this, derivatives and guards need to be written in a particular syntax. This can complicate interpreter writing, for example, to call a sinusoidal function in dReach, using sin(x) would suffice whereas HyCreate would use Math.sin(x).

As an example of the complications, in dReach, a derivative is written as shown

in Listing 6.2. On the other hand, the same derivative for HyCreate is shown in Listing 6.3.

Listing 6.2: A derivative in dReach

```
1 d/dt[v] = ((((r-v)*1)/1)-((v*0.2)/1));
```

Listing 6.3: A derivative in HyCreate

```
1 return new Interval(((((($R-$V)*1)/1)-(($V*0.2)/1)),
2         ((((($R-$V)*1)/1)-(($V*0.2)/1))));
```

Handling the syntax in one particular language is not very difficult, but generalizing a method for all tools is a challenging problem. This is especially true when defining specific properties in one tool that have similar or no use in another. A single interpreter has been written to generate code for both tools described. Normally an interpreter can directly translate models into artifacts, however here a model is first translated into a data model in the JavaScript Object Notation (JSON) format. The structure of the data model closely represents the metamodel, but has added attributes for easier model traversal. For example, a mode under the specification of WebGME or HyCreate has no information on the incoming and outgoing transitions, which would be a helpful part of generating a dReach configuration file. Likewise, only modes have transitions under the dReach model, so iterating over just transitions for HyCreate file generation is a challenge. The data model has all components flattened for easy iteration, and each component has references to all connected components. Each mode has a list of incoming and outgoing transitions, and transitions are contained under the diagram with references to the source and destination.

This data model acts as an intermediary step so that the artifact generation becomes trivial for both tools. The data model is also populated with elements

that are tool-specific, such as the faceSizeRatio and GridSize needed for HyCreate, and the unique integers for each mode needed for dReach. Math expressions for derivatives are also generated for inclusion in the data model. Embedded javascript may then be used to fill out a template version of a configuration file for each tool. A data model snippet of a derivative in a mode is shown in JSON format where differences in expressions and particular tool-specific attributes are shown in Listing 6.4.

Listing 6.4: A full derivative definition in a data model

```
"v_eNd": {
    "name": "v-derivative",
    "uname": "v-derivative_emRG",
    "expression": "((((r_eNc-v_eNd)*1)/1)-((v_eNd*0.2)/1))",
    "id": "/e/m/R/G",
    "faceSizeRatio": 0.1,
    "gridSize": 0.02,
    "regridRatio": 1.7,
    "expressionHyCreate": "((((($R-$V)*1)/1)-(($V*0.2)/1))",
    "expressionDReach": "((((r-v)*1)/1)-((v*0.2)/1))",
    "derivativeAssignment": "v_eNd"
},
```

The practice of transforming a model to fit two different verification tools shows how further verification tools could be more quickly implemented. Production of a flat data model could be in a generalized form for a majority of metamodels. Verification tools would then only need to be a skeleton version. For the purposes of this particular hybrid design language, only one verification tool is needed to assess reachability. HyCreate was chosen based purely on ease of execution, though switching to dReach in the future will be easier for backend server integration.

6.1.5 Expert Block for Reachability

The expert block for the constraint of reaching a particular set of coordinates within a radius is based on the reachability information provided by HyCreate. A final reachable set that is not within the target circle needs to have a modification that could occur either with a change in gain tuning, a transition time, or an additional mode. This expert takes the approach of adding a control mode at the end of the control sequence. Figure 6.8 shows a simple example illustrated on data provided by HyCreate. The first model iteration only has a single control mode, driving the vehicle along the x-axis. This does not achieve the target circle. The expert block, in this case, takes the centroid of the final reachable set and solves the tangent circular path needed to reach the goal center. The path information is then used to set the controller set points in a new mode. Invocation of the model transformation shown in Figure 6.5 then modified the model, shown as 2 mode solution. This the new mode, the HyCreate tool is invoked once more and demonstrates a proper reachable set.

The kinematic bicycle model provides a simple geometric solution for reaching the goal from a current state. This can be solved by first determining the centroid of the current state from the verification output, for the position (x, y) and heading ϕ. To make the solution easier, the goal point (x_g, y_g) is transformed to be with respect to the current heading and location, as in Equations 6.3.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\phi) & sin(\phi) \\ -sin(\phi) & cos(\phi) \end{bmatrix} \begin{bmatrix} x_g - x \\ y_g - y \end{bmatrix} \tag{6.3}$$

After this transformation the trajectory circular sector angle ψ can be determined, as in Equation 6.4.

$$\psi = 2tan^{-1}(\frac{x'}{y'}) \tag{6.4}$$

From knowing the sector angle, the turning radius r of the final motion can be

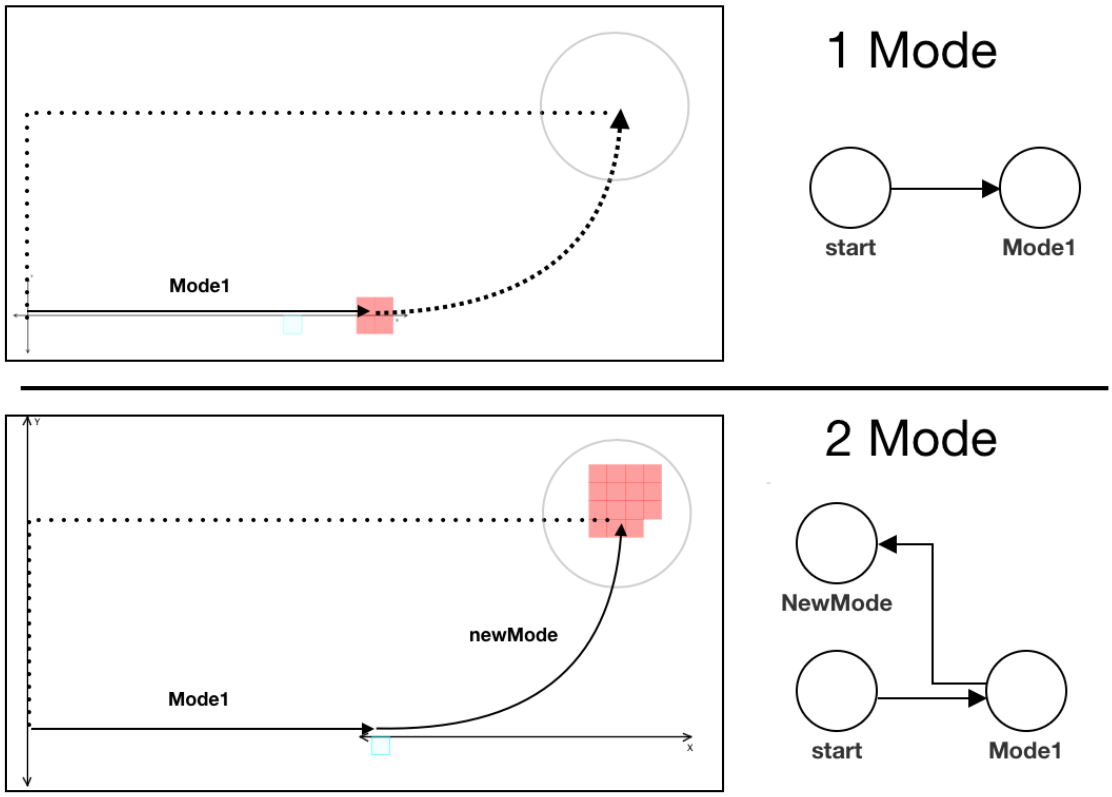

Figure 6.8: Output of HyCreate with a single mode system, marked to show the unreached circle. A second system is shown after model transformation with the necessary added mode to reach the goal coordinates.

determined.

$$r = \frac{\sqrt{x'^2 + y'^2}}{sin(\psi)} sin(\frac{\pi}{2} - \frac{\psi}{2}) \tag{6.5}$$

The turning radius can then be used to determine the front wheel angle θ, bases on the wheelbase L.

$$\theta = tan^{-1}(\frac{L}{r}) \tag{6.6}$$

The only other two values to set are the speed and the transition time. Since only the traversal distance is the most important part, one value can be set and the other can be calculated. for this example, the speed is set as $v = 1$, and the transition time t can be determined by the arc length.

$$t = r\psi \tag{6.7}$$

For this particular language, only simple solutions to the expert blocks were introduced. Though this simple case can be solved by this expert, it is clear that the reachable set became larger in area. For a more complicated system with many modes, this may grow large enough to break feasibility of this solution. Instead a reduction in modes and gain tweaking may be a better approach. This solution is here to illustrate a simple method of model evolution to correct a model.

6.1.6 Expert Block for LTI

There are two designed expert blocks for the simple hybrid language to handle the three possible constraint violations. The first two constraint violations of rise time and overshoot are handled by the same expert block involving controller tuning. The controller tuner is provided with constraint violations and then makes adjustments to the corresponding controller gains. As an example, a user may define the rise time constraint to be 0.4 seconds but must never tune the gains too high to create overshoot. From the previous example, we see that a small amount of overshoot

Iteration	Gain	Overshoot
1	100	0.6962
2	90	0.3228
3	81	0.1054
4	72.9	0.0127
5	65.61	0

Table 6.2: Gain tuning from overshoot violation

Iteration	Gain	RiseTime
1	10	0.7693
2	14	0.7045
3	19.6	0.6284
4	38.416	0.4531
5	53.7824	0.3636

Table 6.3: Gain tuning from rise time violation

does exist, resulting in a constraint violation. We also see that the rise time has not exceeded the user's defined constraint. A heuristic-based expert block may then have a rule to decrease the gain by 10%, then try again. Using this rule, table 6.2 shows how the dynamics of the model evolve over each iteration.

This heuristic appears to work well for overshoot violations. Likewise, if the gain is tuned to be too small, then the rise time will be too slow. If we start with an initial model with a gain of 10, then the rise time will be 0.7693. Similar to the overshoot handling, the expert may be designed to increase the gain by 40% until the rise time does not violate the constraint. This rule results in the iterations shown in table 6.3. Both the rise time and overshoot corrections can be codified as in Equation 6.8. K_P represents the proportional gain of the controller to be tuned.

$$K_P \leftarrow K_P * \begin{cases} 0.9 & overshoot > 0 \\ 1.4 & risetime > 0.4 \end{cases} \tag{6.8}$$

6.1.7 DCF as a Control System

For the sake of example, take the previous expert block for LTI constraint violations and rework it to more closely represent a control system by correcting the model based on the magnitude away from being constraint violation-free. This is similar to what a proportional gain controller accomplishes. Equation 6.8 has now been update to reflect this expert block design change, as shown in Equation 6.9.

$$K_P \leftarrow K_P - K_{design} \begin{cases} -(0 - overshoot) & overshoot > 0 \\ 0.4 - risetime & risetime > 0.4 \end{cases} \tag{6.9}$$

K_{design} now represents a proportional change to the design based on constraint violation error, adjusting the rate of change of the model's proportional gain K_P. With this new expert in place, different behaviors may be observed with different values for K_{design}. Assuming an initial gain of $K_P = 100$ as before, the resulting trend can be seen in Figure 6.9. With these parameters, it can be observed that the controller starts with an overshoot of 0.7, then exponentially decays over each iteration. At iteration 25, the overshoot is very close to 0. These characteristics are similar to an over-dampened controller.

For the next example, K_{design} has been increased to 80, to be more aggressive and hopefully reduce the number of iterations. Figure 6.10 shows the results with $K_{design} = 80$. At first glance, this also appears the be over-dampened but this time in rise time. However, with the same initial conditions, the first iteration shows to be the same overshoot as before, at 0.7. The correction for this overshoot error was so extreme that the next iteration failed the rise time constraint. This example, therefore, represents an overshoot in design.

Since moving from $K_{design} = 20$ to $K_{design} = 80$ went from an over-dampened to an under-dampened system, heuristically it may make sense to attempt a middle ground, say $K_{design} = 50$. Doing so corrects the model in a single iteration, resulting in 0 overshoot and a rise time of 0.3155.

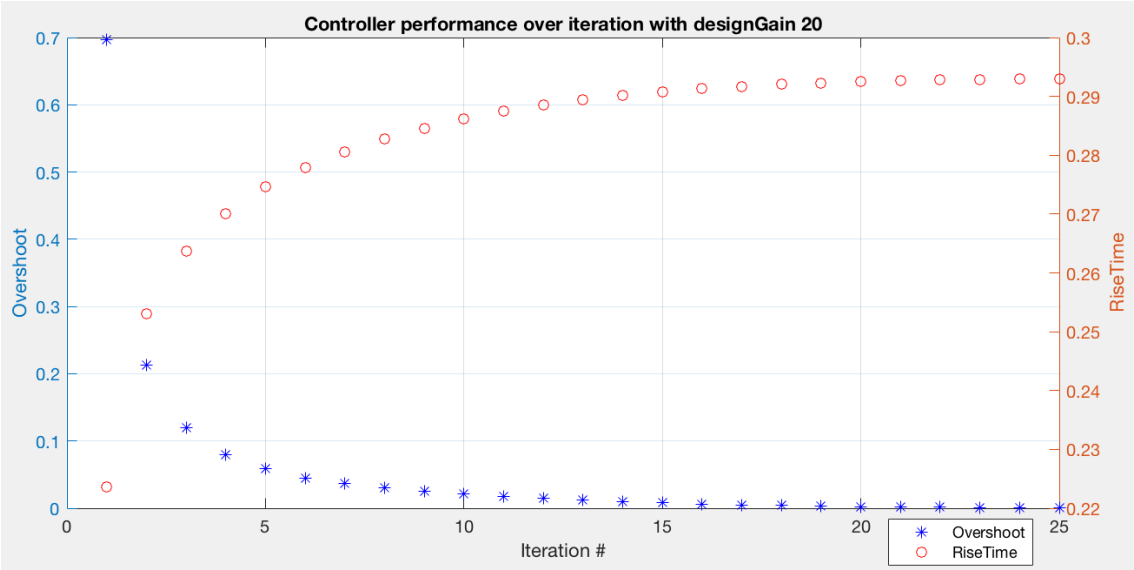

Figure 6.9: Controller performance per iteration of model correction with $K_{design} = 20$, representing an over-dampened system.

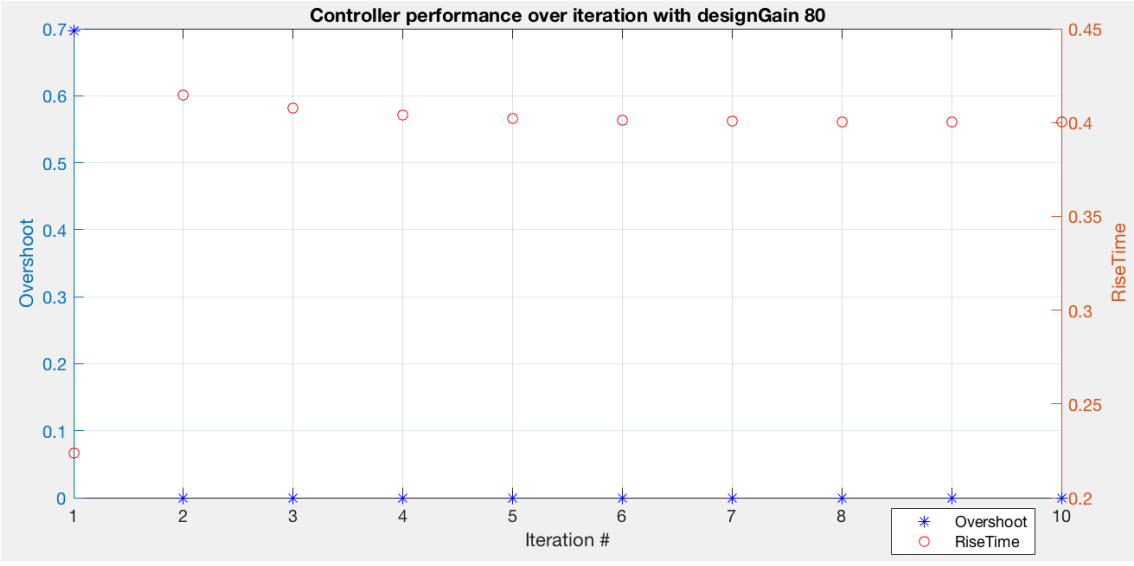

Figure 6.10: Controller performance per iteration of model correction with $K_{design} = 80$, representing an overshoot.

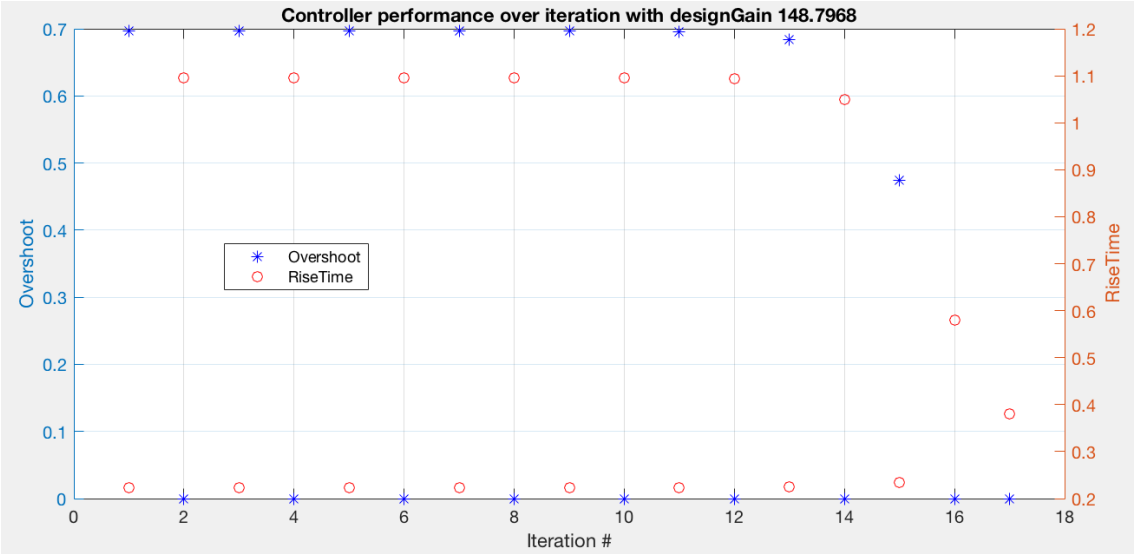

Figure 6.11: Controller performance per iteration of model correction with $K_{design} \approx 148.7968$, closely representing an oscillator.

Regarding further dynamic behavior of control systems, there exists the construct of having imaginary poles with little or no real number component. In such cases, oscillatory behavior can be observed. Since an overshoot example from increasing K_{design} has already been shown, this suggests that oscillatory may be observed by further increasing K_{design}. With $K_{design} = 148.796828025$, oscillatory behavior appears, as shown in Figure 6.11. This does not represent a pure oscillator since the oscillation eventually dies down and converges on a corrected model at the 17th iteration, however it does exhibit similar behavior to a dynamical system with imaginary poles with a little real part. Every other iteration, the behavior of the controller represents a close-to-previous yet closer-to-correct result, which is why the model eventually meets the dynamic constraints.

The value of K_{design} for the previous example was discovered by brute force attempts to create long oscillations. By increasing this value such that $K_{design} = 148.7969$, the design system is unstable. Figure 6.12 shows the results of the unstable design expert. In this example, every other iteration has the behavior of being close-to-previous yet further-from-correct. At iteration 12, stepinfo resulted in producing

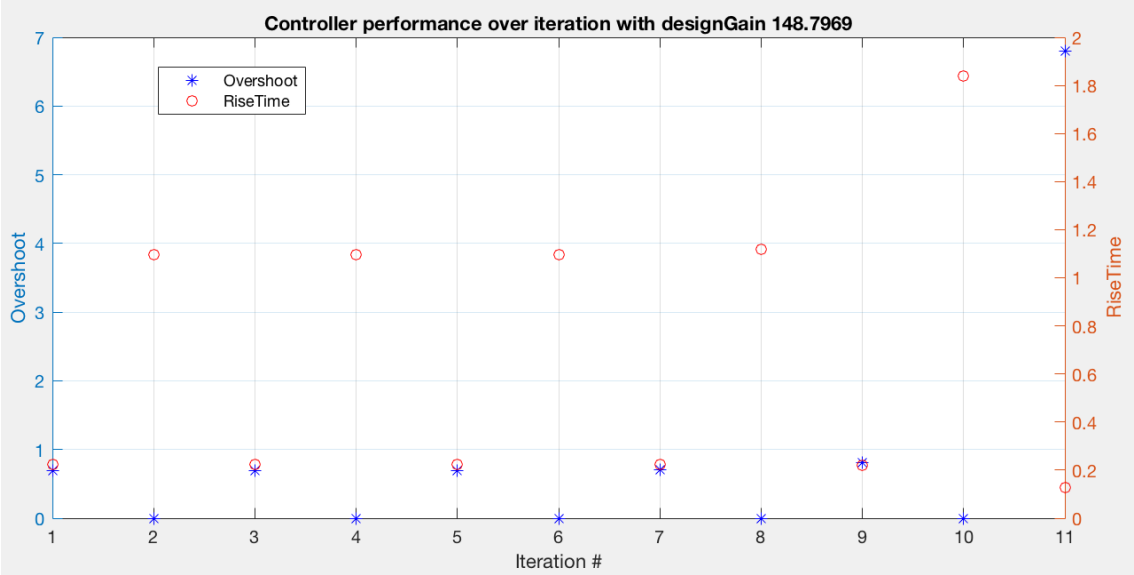

Figure 6.12: Controller performance per iteration of model correction with $K_{design} = 148.7969$, closely representing an unstable system

NaN as verification output, which is due to the expert block setting $K_P \approx -814.5$. This represents a positive feedback loop, and stepinfo can only produce numerical verification output for stable systems.

CHAPTER 7

Conclusion

7.1 Contribution

The DCFML provides an extension to WebGME's meta-metamodel to consider DCF integration in DSML design. Implementing automatic verification methods in CPS based DSMLs can be a critical part of ensuring that models abide by important safety constraints. The verification loop can also be potentially fully closed to automatically correct constraint violations through model evolution. This framework aids a DSML designer by extending an already familiar metamodel with the methods needed for model transformation, constraint modeling, and expert block connectivity. By providing a concrete syntax for DCF based DSMLs, unrestricted implementation can be avoided and therefore aid in dynamic requirement validation and verification. Implementation of DCF is not a new idea, but the abstraction as an extension to the metamodel is new.

Design of the DCFML also provides a concrete example regarding the improvement of DSML design in general. Outside of implementing DCF, DSML design processes may benefit from thinking about constraints imposed on the metamodeling process, through abstract thinking of the meta-metamodel. The DCFML example shows that a family of languages for verification-important CPS domains benefited from rethinking the meta-metamodel. The same could be true for further families of languages in CPSs, not necessarily with a focus on verification but with other potential qualities. These may even extend outside of CPS development and impact DSML design in general.

This work briefly looked at case studies implementing the framework into both

a trajectory language and a simple hybrid controller design language. The hybrid controller DSML demonstrated the design of the metamodel, model transformation, expert blocks, and verification tool integration. Interpretation of the design automatically fit constraints into the metamodel, and generated code for model transformations. Expert blocks were also constructed to demonstrate simple methods of closing the loop. These are primitive examples but demonstrations were shown on how models can be automatically evolved into working solutions.

7.1.1 Limitations

The DCFML described is in its infancy, and parts of the design could be further defined. The expert block is lacking in its current state, requiring code to be written at a low level to be properly implemented. As discussed, expert blocks can vary greatly so abstraction for general purpose modeling is not a small task. Doing so could result in coupling model transformations with the expert block during system interpretation, greatly reducing the time of implementing DCFML. Implementing expert block design in such a language is challenging due to the algorithmic nature, and could easily become too generic to be useful in a visual modeling scenario.

Verification tools also vary greatly by their necessary inputs and outputs. Generation of a model interpreter to create a flat metamodel could greatly reduce implementation time. This way a DSML designer would only need to focus on building a template of verification tools rather than build custom transformations to produce verification tool artifacts. This is again a challenging task since the particular DSML to be designed may not fit under a general structure.

Dynamic constraints are also under-modeled in the DCFML. Defining simple data types is the only supported format. Cases of requirements exist like avoiding a particular area for reachability feedback, which cannot be expressed in the current form. This also ties in with the comparison operation and verification output modeling. All of this needs to be handwritten like the components of the DCFML listed

above.

One of the keys to the success of the DCFML is adapting prior metamodels to fit under the new construct. As presented, the DCFML is only designed for WebGME, and other DSMEs are not supported. Theoretically, it should be workable in other DSMEs, however there could be some metamodels that cannot fully self define, which is a necessary part in establishing the DCFML. Also, some modeling languages like Simulink do not have metamodeling methods, and therefore the DCFML cannot be implemented.

7.2 Future Work

The DCFML is a prime candidate for future work considering the enormous impact it could have on both model design and DSML design. Future work can be determined based on the observation of current limitations. Also, future analysis may be performed that has not yet been presented. Such analysis may be explored by looking at further domain parallels between the DCFML and control system design.

7.2.1 Improving Limitations

Since the DCFML aims to abstract the dynamic constraints as part of closing the loop, working to further refine dynamic constraint modeling techniques would be highly beneficial. The process of metamodeling enforces the structural constraints during model creation. Further structural constraints have also been imposed by specifically creating languages to add rich rulesets to the structure, such as the Object Constraint Language (OCL) [112]. OCL lets constraints be imposed on model design based on the model itself, as opposed to just the formal specification. Similarly, dynamic constraints enforce corrections based on the model beyond the metamodel. Having a formalized Dynamic Constraint Language (DCL) could be directly interfaced with, and improve the DCFML.

Current efforts on the CPS-VO Active Resources involve working on the democratization of verification tools through the use of modeling languages. Their method focuses on the evaluation of verification tools for CPS applications. These focus on the modeling techniques needed for full verification tool use rather than for a DSML. Such efforts could eventually establish standardized interfaces for verification tools that are within similar domains. The examples presented in this work involved assessing two different verification tool input modeling techniques. Having a standardized interface for verification tool types could act as interfaces for DSMLs.

The limitations may be solved from the implementation of the DCFML into further domains. Each DSML provides varying constraints, verification, and design techniques. Implementing more DSML case studies into the DCFML will help carve out the modeling needs for general-purpose use.

7.2.2 Designability and Verifiability

In control systems theory, there exists the potential to have an under-actuated system. Under-actuation means that the system cannot be commanded to follow arbitrary trajectories. A simple example of this scenario is when a system has fewer actuators than it has degrees of freedom. This does not however mean that desired states cannot be reached. This term for determining whether an under-actuated can be commanded is controllability.

The duality to this concept deals with an under-sensed system. Similarly, under-sensed systems do not provide the sensors necessary to measure all system states. In other words, the number of sensors is less than the degrees of freedom. This study is called observability.

Since DCF involves a control system structure and properties of automatic correction performance have been shown, concepts of controllability and observability may also be related to DCF. To have more clarity on the relationship between domains, controllability could be known as designability, and observability could be

known as verifiability. Designability and verifiability may be similar in concept, but may not have a direct relationship. They do however relate from a high-level perspective.

Designability deals with the ability to evolve the model into all possible solution states. A designable DCF system is one that can translate dynamic constraint violations into corrected models. Establishing designability may be of critical importance in validating full automatic correction. In control systems, there are usually constraints on the physical system that makes full articulation impossible. Contrary, model evolution is only limited by the model transformations and metamodel. Determining the metrics of what is designable could involve future studies.

Similarly, verifiability would involve the ability to understand the full dynamic behavior of a system. A verifiably DCF would be able to fully map all models into dynamic behavior for requirement comparison. Establishing verifiability may be critical to having full automatic correction. Again, control systems are usually constrained by the sensing technology or physical implementation. DCF will be limited both by the availability of verification method and the semantic mapping between models and verification tool input. Verifiability could involve another set of metrics that could be studied in the future.

APPENDIX A

Generated Transformation Code

A.1 Generated Transformation Code

The following is the function generated from the AddModeAndTransformation transformation in figure 6.5. This code is intended to be implemented in a Javascript-based plugin in WebGME. The Core API is responsible for performing the actual modification. The transformation model is used to determine the inputs for this function, in this case being references by WebGME generated IDs.

Listing A.1: Generated transformation from AddModeAndTransformation.

```
1  HybridLanguage.prototype.addModeRule =
2      function (dataModel, DiagramMMReferenceID,
3                      ModeMMReferenceID) {
4          var self = this;
5
6          var newTransitionMeta = 'Transition';
7          var NewModeMeta = 'Mode';
8
9          var DiagramMMReferenceNode =
10                 self.pathToNode[DiagramMMReferenceID];
11         var ModeMMReferenceNode =
12                 self.pathToNode[ModeMMReferenceID];
13
14         var paramsnewTransition = {
15             parent: DiagramMMReferenceNode,
```

```
16              base: self.META[newTransitionMeta]
17          };
18          var newTransition =
19                  self.core.createNode(paramsnewTransition);
20          self.core.setAttribute(newTransition, 'name',
21                                 'newTransition');
22
23          var paramsNewMode = {
24              parent: DiagramMMReferenceNode,
25              base: self.META[NewModeMeta]
26          };
27          var NewMode = self.core.createNode(paramsNewMode);
28          self.core.setAttribute(newMode, 'name', 'NewMode');
29
30          self.core.setPointer(newTransition, 'src',
31                               ModeMMReferenceNode);
32          self.core.setPointer(newTransition, 'dst',
33                               NewMode);
34
35          self.core.setAttribute(NewMode, 'SetPointAngle',
36                                 0.0);
37          self.core.setAttribute(NewMode, 'SetPointSpeed',
38                                 1.0);
39  };
```

Modifying a model's structure is much more difficult compared to modifying attributes. The following function is a demonstration of modifying a node's value based on the node id, attribute name, and attribute value.

Listing A.2: Generated transformation from ModifyControllerValue.

```
1 HybridLanguage.prototype.setAttribute =
2     function (dataModel, MMReferenceID, attributeName,
3                     attributeValue) {
4         var self = this;
5         var Node = self.pathToNode[MMReferenceID];
6
7         self.core.setAttribute(Node, attributeName,
8                                     attributeValue);
9 };
```

REFERENCES

[1] A. Agrawal. Graph rewriting and transformation (great): a solution for the model integrated computing (mic) bottleneck. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 364–368. IEEE, 2003.

[2] H. Albin-Amiot and Y.-G. Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[3] M. Althoff. An introduction to cora 2015. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

[4] R. E. Andersen, E. B. Hansen, D. Cerny, S. Madsen, B. Pulendralingam, S. Bøgh, and D. Chrysostomou. Integration of a skill-based collaborative mobile robot in a smart cyber-physical environment. *Procedia Manufacturing*, 11:114–123, 2017.

[5] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[6] P. Antsaklis. Goals and challenges in cyber-physical systems research editorial of the editor in chief. *IEEE Transactions on Automatic Control*, 59(12):3117–3119, 2014.

[7] K. J. Åström and T. Hägglund. Revisiting the ziegler–nichols step response method for pid control. *Journal of process control*, 14(6):635–650, 2004.

[8] K. J. Åström, T. Hägglund, C. C. Hang, and W. K. Ho. Automatic tuning and adaptation for pid controllers-a survey. *Control Engineering Practice*, 1(4):699–714, 1993.

[9] M. W. Aziz and M. Rashid. Domain specific modeling language for cyber physical systems. In *2016 International Conference on Information Systems Engineering (ICISE)*, pages 29–33. IEEE, 2016.

[10] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. F. R. Jesus, R. F. Berriel, T. M. Paixão, F. Mutz, et al. Self-driving cars: A survey. *arXiv preprint arXiv:1901.04407*, 2019.

[11] R. Baheti and H. Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[12] S. Bak. Hycreate: A tool for overapproximating reachability of hybrid automata. *Retrieved January*, 17:2016, 2013.

[13] B. Balaji, A. Faruque, M. Abdullah, N. Dutt, R. Gupta, and Y. Agarwal. Models, abstractions, and architectures: The missing links in cyber-physical systems. In *Proceedings of the 52nd Annual Design Automation Conference*, page 82. ACM, 2015.

[14] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*, 1, 2007.

[15] J. Barnat, L. Brim, and J. Stříbrná. Distributed ltl model-checking in spin. In *International SPIN Workshop on Model Checking of Software*, pages 200–216. Springer, 2001.

[16] J. Bézivin and O. Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001.

[17] J. Bézivin, F. Jouault, and J. Paliès. Towards model transformation design patterns. In *Proceedings of the First European Workshop on Model Transformations (EWMT 2005)*, 2005.

[18] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl. Multi-domain modeling of cyber-physical systems using architectural views. 2010.

[19] R. Bucher and S. Balemi. Rapid controller prototyping with matlab/simulink and linux. *Control Engineering Practice*, 14(2):185–192, 2006.

[20] M. Buehler, K. Iagnemma, and S. Singh. *The 2005 DARPA grand challenge: the great robot race*, volume 36. Springer, 2007.

[21] M. Bunting, Y. Zeleke, K. McKeever, and J. Sprinkle. A safe autonomous vehicle trajectory domain specific modeling language for non-expert development. In *Proceedings of the International Workshop on Domain-Specific Modeling*, pages 42–48. ACM, 2016.

[22] J. Cabot, R. Clarisó, E. Guerra, and J. De Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[23] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

[24] H. Cho and J. Gray. Design patterns for metamodels. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 25–32. ACM, 2011.

[25] A. Christoph. Describing horizontal model transformations with graph rewriting rules. In *Model Driven Architecture*, pages 93–107. Springer, 2004.

[26] M. H. Cintuglu, O. A. Mohammed, K. Akkaya, and A. S. Uluagac. A survey on smart grid cyber-physical system testbeds. *IEEE Communications Surveys and Tutorials*, 19(1):446–464, 2017.

[27] T. Clark, A. Evans, S. Kent, and P. Sammut. The mmf approach to engineering object-oriented design languages. 2001.

[28] M. F. Costabile, D. Fogli, P. Mussio, and A. Piccinno. Visual interactive systems for end-user development: a model-based design methodology. *IEEE transactions on systems, man, and cybernetics-part a: systems and humans*, 37(6):1029–1046, 2007.

[29] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[30] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[31] L. C. B. da Silva, R. M. Bernardo, H. A. de Oliveira, and P. F. Rosa. Multi-uav agent-based coordination for persistent surveillance with dynamic priorities. In *Military Technologies (ICMT), 2017 International Conference on*, pages 765–771. IEEE, 2017.

[32] J. R. Davis. Model integrated computing : A framework for creating domain specific design environments. 2002.

[33] P. Deng, F. Cremona, Q. Zhu, M. Di Natale, and H. Zeng. A model-based synthesis flow for automotive cps. In *Proceedings of the ACM/IEEE Sixth*

International Conference on Cyber-Physical Systems, pages 198–207. ACM, 2015.

[34] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[35] L. dos Santos Coelho. Tuning of pid controller for an automatic regulator voltage system using chaotic optimization approach. *Chaos, Solitons & Fractals*, 39(4):1504–1514, 2009.

[36] K. Duddy, A. Gerber, M. Lawley, K. Raymond, and J. Steel. Model transformation: A declarative, reusable patterns approach. In *Seventh IEEE International Enterprise Distributed Object Computing Conference, 2003. Proceedings.*, pages 174–185. IEEE, 2003.

[37] M. J. Emerson, J. Sztipanovits, and T. Bapty. A mof-based metamodeling environment. *J. UCS*, 10(10):1357–1382, 2004.

[38] J. H. Frazer, J. M. Frazer, X. Liu, M. X. Tang, and P. Janssen. Generative and evolutionary techniques for building envelope design. 2002.

[39] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, pages 379–395. Springer, 2011.

[40] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[41] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–387. Springer, 2011.

[42] C. Gerking, W. Schäfer, S. Dziwok, and C. Heinzemann. Domain-specific model checking for cyber-physical systems. In *MoDeVVa@ MoDELS*, pages 18–27, 2015.

[43] J. Gray, S. Neema, J.-P. Tolvanen, A. S. Gokhale, S. Kelly, and J. Sprinkle. Domain-specific modeling. *Handbook of dynamic system modeling*, 7:7–1, 2007.

[44] M. Heß, M. Kaczmarek, U. Frank, L. Podleska, and G. Täger. A domain-specific modelling language for clinical pathways in the realm of multi-perspective hospital modelling. In *ECIS*, 2015.

[45] G. Horlick, R. Lawson, D. Morgan, P. Musto, J. Smedley, K. Wadland, R. Woodward, S. Bergan, and W. M. Katz. User-guided autorouting, May 19 2009. US Patent 7,536,665.

[46] S.-J. Huang and H.-Y. Chen. Adaptive sliding controller with self-tuning fuzzy compensation for vehicle suspension control. *Mechatronics*, 16(10):607–622, 2006.

[47] M.-E. Iacob, M. W. Steen, and L. Heerink. Reusable model transformation patterns. In *2008 12th Enterprise Distributed Object Computing Conference Workshops*, pages 1–10. IEEE, 2008.

[48] H. I. Ismail, I. V. Bessa, L. C. Cordeiro, E. B. de Lima Filho, and J. E. Chaves Filho. Dsverifier: A bounded model checking tool for digital systems. In *International SPIN Workshop on Model Checking of Software*, pages 126–131. Springer, 2015.

[49] E. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software & Systems Modeling*, 8(4):451–478, 2009.

[50] E. K. Jackson and J. Sztipanovits. Towards a formal foundation for domain specific modeling languages. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 53–62. ACM, 2006.

[51] D. Jia, K. Lu, J. Wang, X. Zhang, and X. Shen. A survey on platoon-based vehicular cyber-physical systems. *IEEE communications surveys & tutorials*, 18(1):263–284, 2016.

[52] Y. Jiang, H. Song, R. Wang, M. Gu, J. Sun, and L. Sha. Data-centered runtime verification of wireless medical cyber-physical system. *IEEE transactions on industrial informatics*, 13(4):1900–1909, 2016.

[53] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[54] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[55] P. A. Judas and L. E. Prokop. A historical compilation of software metrics with applicability to nasa's orion spacecraft flight software sizing. *Innovations in Systems and Software Engineering*, 7(3):161–170, 2011.

[56] T. Kecskes, P. Meijer, T. T. Johnson, and M. Lucas. a design studio for verification tools. In *Proceedings of the Workshop on Design Automation for CPS and IoT*, pages 60–61. ACM, 2019.

[57] F. Khorrami, P. Krishnamurthy, and R. Karri. Cybersecurity for control systems: A process-aware perspective. *IEEE Design & Test*, 33(5):75–83, 2016.

[58] S. Khwaja and M. Alshayeb. Survey on software design-pattern specification languages. *ACM Computing Surveys (CSUR)*, 49(1):21, 2016.

[59] J. Knight, J. Xiang, and K. Sullivan. A rigorous definition of cyber-physical systems. *Trustworthy Cyber-Physical Systems Engineering*, 47:47–70, 2016.

[60] N. P. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IROS*, volume 4, pages 2149–2154. Citeseer, 2004.

[61] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1094–1099. IEEE, 2015.

[62] S. Kong, S. Gao, W. Chen, and E. Clarke. dreach: δ-reachability analysis for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 200–205. Springer, 2015.

[63] S. Krish. A practical generative design method. *Computer-Aided Design*, 43(1):88–100, 2011.

[64] K. Lano and S. Kolahdouz-Rahimi. Model-transformation design patterns. *IEEE Transactions on Software Engineering*, 40(12):1224–1259, 2014.

[65] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[66] P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, et al. Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6. IEEE, 2016.

[67] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environ-

ment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, page 1, 2001.

[68] E. A. Lee. Cyber physical systems: Design challenges. In *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[69] B. Lewis. Position paper : Need for architecture description language with standardized well defined meaning for architecture centric engineering of cyber-physical systems. 2010.

[70] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5 flight 501 failure report by the inquiry board, 1996.

[71] S. Liu, J. Tang, C. Wang, Q. Wang, and J.-L. Gaudiot. A unified cloud platform for autonomous driving. *Computer*, 50(12):42–49, 2017.

[72] J. Lopes and A. Leitão. Portable generative design for cad applications. 2011.

[73] C. Lv, Y. Liu, X. Hu, H. Guo, D. Cao, and F.-Y. Wang. Simultaneous observation of hybrid states for cyber-physical systems: A case study of electric vehicle powertrain. *IEEE Transactions on Cybernetics*, 48(8):2357–2367, 2018.

[74] J. Lygeros, C. Tomlin, and S. Sastry. Controllers for reachability specifications for hybrid systems. *Automatica*, 35(3):349–370, 1999.

[75] T. T. Mac, C. Copot, T. T. Duc, and R. De Keyser. Ar. drone uav control parameters tuning based on particle swarm optimization algorithm. In *2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–6. IEEE, 2016.

[76] 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.

[77] P. Martinet, C. Thibaud, B. Thuilot, and J. Gallice. Robust controller synthesis in automatic guided vehicles applications. *Advances in Vehicles Control and Safety (AVCS'98)*, pages 395–401, 1998.

[78] D. Massey. Applying cybersecurity challenges to medical and vehicular cyber physical systems. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, pages 39–39. ACM, 2017.

[79] 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*, pages 17–22. ACM, 2015.

[80] W. Meng, J. Park, O. Sokolsky, S. Weirich, and I. Lee. Verified ros-based deployment of platform-independent control systems. In *NASA Formal Methods Symposium*, pages 248–262. Springer, 2015.

[81] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[82] I. M. Mitchell. A toolbox of level set methods. *UBC Department of Computer Science Technical Report TR-2007-11*, 2007.

[83] R. Mitchell and R. Chen. Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. *IEEE Transactions on Dependable and Secure Computing*, 12(1):16–30, 2015.

[84] K. Mitzner. *Complete PCB design using OrCad capture and layout*. Elsevier, 2011.

[85] H. Neema, P. Volgyesi, B. Potteiger, W. Emfinger, X. Koutsoukos, G. Karsai, Y. Vorobeychik, and J. Sztipanovits. Sure: An experimentation and evaluation testbed for cps security and resilience: Demo abstract. In *Proceedings of the 7th International Conference on Cyber-Physical Systems*, page 27. IEEE Press, 2016.

[86] C.-P. S. V. Organization. Verification tool library.

[87] F. Paterno. *Model-based design and evaluation of interactive applications*. Springer Science & Business Media, 1999.

[88] S. Petti and T. Fraichard. Safe motion planning in dynamic environments. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2210–2215. IEEE, 2005.

[89] P. Polack, F. Altché, B. d'Andréa Novel, and A. de La Fortelle. The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles? In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 812–818. IEEE, 2017.

[90] S. M. Pradhan, A. Dubey, A. Gokhale, and M. Lehofer. Chariot: A domain specific language for extensible cyber-physical systems. In *Proceedings of the workshop on domain-specific modeling*, pages 9–16. ACM, 2015.

[91] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[92] D. Redman, D. Ward, J. Chilenski, and G. Pollari. Virtual integration for improved system design. In *First Analytic Virtual Integration of Cyber-Physical Systems Workshop*, pages 57–64, 2010.

[93] S. A. Rees, T. Kecskes, P. Meijer, T. T. Johnson, K. Dey, P. Tabuada, and M. Lucas. Cyber-physical systems virtual organization: Active resources: enabling reproducibility, improving accessibility, and lowering the barrier to entry. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 340–341. ACM, 2019.

[94] K. Sampigethaya and R. Poovendran. Aviation cyber–physical systems: Foundations for future aircraft and air transport. *Proceedings of the IEEE*, 101(8):1834–1855, 2013.

[95] B. J. Sauser, R. R. Reilly, and A. J. Shenhar. Why projects fail? how contingency theory can provide new insights–a comparative analysis of nasa's mars climate orbiter loss. *International Journal of Project Management*, 27(7):665–679, 2009.

[96] T. S. Schei. A method for closed loop automatic tuning of pid controllers. *Automatica*, 28(3):587–591, 1992.

[97] J. Schroeder, C. Berger, A. Knauss, H. Preenja, M. Ali, M. Staron, and T. Herpel. Predicting and evaluating software model growth in the automotive industry. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 584–593. IEEE, 2017.

[98] K. Shea, R. Aish, and M. Gourtovaia. Towards integrated performance-driven generative design tools. *Automation in Construction*, 14(2):253–264, 2005.

[99] J. Shi, J. Wan, H. Yan, and H. Suo. A survey of cyber-physical systems. In *2011 international conference on wireless communications and signal processing (WCSP)*, pages 1–6. IEEE, 2011.

[100] C. A. Smith and A. B. Corripio. *Principles and practice of automatic process control*, volume 2. Wiley New York, 1985.

[101] K. Smolander, K. Lyytinen, V.-P. Tahvanainen, and P. Marttiin. Metaedit-a flexible graphical environment for methodology modelling. In *International Conference on Advanced Information Systems Engineering*, pages 168–193. Springer, 1991.

[102] D. Sonntag, S. Zillner, P. van der Smagt, and A. Lörincz. Overview of the cps for smart factories project: Deep learning, knowledge acquisition, anomaly detection and intelligent user interfaces. In *Industrial internet of things*, pages 487–504. Springer, 2017.

[103] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[104] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, and E. Jackson. Openmeta: A model-and component-based design tool chain for cyber-physical systems. In *From Programs to Systems. The Systems perspective in Computing*, pages 235–248. Springer, 2014.

[105] J. Sztipanovits, T. Bapty, S. Neema, X. Koutsoukos, and E. Jackson. Design tool chain for cyber-physical systems: Lessons learned. In *Proceedings of the 52nd Annual Design Automation Conference*, page 81. ACM, 2015.

[106] P. Tabuada and G. J. Pappas. Model checking ltl over controllable linear systems is decidable. In *International Workshop on Hybrid Systems: Computation and Control*, pages 498–513. Springer, 2003.

[107] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[108] P. Tournassoud. Motion planning for a mobile robot with a kinematic constraint. In *French Workshop on Geometry and Robotics*, pages 150–171. Springer, 1988.

[109] J. Wan, A. Canedo, and M. A. Al Faruque. Functional model-based design methodology for automotive cyber-physical systems. *IEEE Systems Journal*, 11(4):2028–2039, 2017.

[110] Q.-G. Wang, T.-H. Lee, H.-W. Fung, Q. Bi, and Y. Zhang. Pid tuning for improved performance. *IEEE Transactions on control systems technology*, 7(4):457–465, 1999.

[111] X. V. Wang, Z. Kemény, J. Váncza, and L. Wang. Human–robot collaborative assembly in cyber-physical production: Classification framework and implementation. *CIRP annals*, 66(1):5–8, 2017.

[112] J. B. Warmer and A. G. Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[113] J. White, D. C. Schmidt, A. Nechypurenko, and E. Wuchner. Introduction to the generic eclipse modeling system. *Eclipse Magazine*, 6:11–18, 2007.

[114] S. Whitsitt. A methodology for mending dynamic constraint violations in cyber physical systems by generating model transformations. 2014.

[115] 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. IEEE, 2013.

[116] S. Whitsitt, J. Sprinkle, and R. Lysecky. Generating model transformations for mending dynamic constraint violations in cyber physical systems. In *Proceedings of the 14th Workshop on Domain-Specific Modeling*, pages 35–40. ACM, 2014.

[117] X. Yu and Y. Xue. Smart grids: A cyber–physical systems perspective. *Proceedings of the IEEE*, 104(5):1058–1070, 2016.

[118] C. Zhang and D. Budgen. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering*, 38(5):1213–1231, 2012.

[119] 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*, pages 1–6. ACM, 2014.

[120] L. Zhang. Modeling automotive cyber physical systems. In *2013 12th International Symposium on Distributed Computing and Applications to Business, Engineering & Science*, pages 71–75. IEEE, 2013.

[121] Y. Zhang, M. Qiu, C.-W. Tsai, M. M. Hassan, and A. Alamri. Health-cps: Healthcare cyber-physical system assisted by cloud and big data. *IEEE Systems Journal*, 11(1):88–95, 2017.