APECS: Polychrony based End-to-End Embedded System Design and Code Synthesis

Matthew E. Anderson

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Sandeep K. Shukla, Chair
Lamine Mili
Alireza Haghighat
Chao Wang
Yi Deng

April 3, 2015
Blacksburg, Virginia

Keywords: AADL, CPS, Model-based code synthesis, correct-by-construction code synthesis, Polychrony, code generators, OSATE, Ocarina

APECS: Polychrony based End-to-End Embedded System Design and Code Synthesis

Matthew E. Anderson

(ABSTRACT)

The development of high integrity embedded systems remains an arduous and error-prone task, despite the efforts by researchers in inventing tools and techniques for design automation. Much of the problem arises from the fact that the semantics of the modeling languages for the various tools, are often distinct, and the semantics gaps are often filled manually through the engineer's understanding of one model or an abstraction. This provides an opportunity for bugs to creep in, other than standardising software engineering errors germane to such complex system engineering. Since embedded systems applications such as avionics, automotive, or industrial automation are safety critical, it is very important to invent tools, and methodologies for safe and reliable system design. Much of the tools, and techniques deal with either the design of embedded platforms (hardware, networking, firmware etc), and software stack separately. The problem of the semantic gap between these two, as well as between models of computation used to capture semantics must be solved in order to design safer embedded systems.

In this dissertation we propose a methodology for the end-to-end modeling and analysis of safety-critical embedded systems. Our approach consists of formal platform modeling, and analysis; formal application modeling; and 'correct-by-construction' code synthesis with the aim of bridging semantic gaps between the various abstractions and models required for the end-to-end system design. While the platform modeling language AADL has formal semantics, and analysis tools for real-time, and performance verification, the application behavior modeling in AADL is weak and part of an annex. In our work, we create the APECS (AADL and Polychrony based Embedded Computing Synthesis) methodology to allow an embedded system design specification all the way from platform architecture and platform components, the real-time behavior, non-functional properties, as well as the application software modeling. Our main contribution is to integrate a polychronous application software modeling language, and synthesis algorithms in order for synthesis of the embedded software running on the target platform, with the required constraints being met. We believe that a polychronous approach is particularly well suited for a multiprocessor/multi-controller distributed platform where different components often operate at independent rates and concurrently. Further, the use of a formal polychronous language will allow for formal validation of the software prior to code generation. We present a prototype framework that implements this approach, which we refer to as the AADL and Polychrony based Embedded Computing System (APECS). Our prototype utilizes an extended version of Ocarina to provide code generation for the AADL model. Our polychronous modeling language is MRICDF. Our prototype extends Ocarina to support software specification in MRICDF and generate multi-threaded software. Additionally, we implement an automated translation from Simulink to MRICDF, allowing designers to benefit from its formal semantics and exploit engineers' familiarity with Simulink tools, and legacy models. We present case studies utilizing APECS to implement safety critical systems both natively in MRICDF and in Simulink through automated translation.

That this work received support from the US Air Force Research Labs Contract (FA-8750-11-1-0042).

Contents

1 Introduction **1**

1.1 Problem Statement 3

1.2 Contributions 5

1.3 Overview 6

2 Model Driven Engineering **8**

2.1 UML 10

2.1.1 SysML 14

2.1.2 MARTE 17

2.2 MATLAB/Simulink 19

2.3 Related Work - Formalizing Simulink 20

2.3.1 Formalizing SIMULINK's Behavioral Semantics 20

2.3.2 Hybrid-Automata based intermediate formats 21

2.3.3 Translating SIMULINK to Formal Languages 23

2.4 AADL 23

2.5 Related Works - Model Analysis and Code Generation Techniques for AADL 29

2.5.1 Ocarina 29

2.5.2 Additional AADL Based Methodologies 31

3 Formal Language Preliminaries **34**

3.1 Formal Languages 34

3.1.1 Synchronous Languages 35

3.1.2 Polychrony 40

4 APECS Methodology **47**

4.1 Motivation 47

4.2 Approach 48

4.3 Toolchain 51

5 Simulink to Polychrony **58**

5.1 Introduction 58

5.2 Type Inference 61

5.3 Clock Inference 65

5.4 Block Translation 68

6 Case Studies **75**

6.1 Elevator 75

6.2 Phasor Measurement Unit 81

7 Conclusions and Future Work **86**

Bibliography **88**

List of Figures

1.1 Model Driven Architecture Approach 2

2.1 UML Diagrams [61] 11
2.2 SysML Diagrams [31] 14
2.3 Elevator Car Block 16
2.4 Elevator Car Requirements 16
2.5 MARTE Architecture [28] 18
2.6 Annex Subclause Usage 28
2.7 Ocarina Frontend 30
2.8 Ocarina Backend 31

3.1 Esterel ABRO 37
3.2 Lustre ABRO 40
3.3 Signal Function 41
3.4 Signal Delay 41
3.5 Signal Undersampling 42
3.6 Signal Undersampling 42
3.7 Signal Merge 42
3.8 Signal Oversampling 42
3.9 Signal ABRO 43
3.10 $BufferActor$ 44
3.11 $SamplerActor$ 44

3.12 *MergeActor* 45
3.13 *FunctionActor* 45

4.1 Vee Life Cycle [30] 48
4.2 Methodology Phases 49
4.3 APECS Design Flow 51
4.4 Associating MRICDF source files with AADL 52
4.5 Ocarina Frontend with MRICDF extensions 53
4.6 Passing Thread Information 54
4.7 Code Distribution 56

5.1 Sim2Em Translation Flow 60
5.2 SIMULINK Type Lattice [66] 61
5.3 Arithmetic Typing Example 64
5.4 MRICDF Adder 70
5.5 MRICDF Difference 70
5.6 Trigger Port Translation 70
5.7 Inserting Triggers into the System 71
5.8 Enable Port Translation 71
5.9 Virtual Connections 72
5.10 Oversampling Composite 73
5.11 Undersampling Composite 74

6.1 Elevator Top Level AADL Model 75
6.2 Floor Call Panel 76
6.3 Elevator Car Control AADL Model 77
6.4 Door System AADL Model 77
6.5 Door State Table 78
6.6 Door Controller State Flow 78
6.7 EmCodeSyn Model of the Door Control Software 79

6.8 MRICDF Process Declaration 80
6.9 MRICDF Clock Tree 80
6.10 Resulting thread groups 81
6.11 Generated OpenDoor Thread 82
6.12 Generated OpenDoor Function Call 82
6.13 Roscoe Public PMU 83
6.14 Simulink PMU 83
6.15 SFunction Type Filter 83
6.16 Roscoe PMU in MRICDF 84
6.17 Test Generation in MRICDF 84
6.18 Manual Implementation of the Test Generator in MRICDF 84
6.19 PMU Subsystem in MRICDF 85
6.20 Manual Implementation of the PMU in MRICDF 85

List of Tables

2.1 Supported Blocks [22] 20

3.1 Esterel Statements 37
3.2 Lustre Statements 39
3.3 Signal Function Operators 41

5.1 Types of Supported Simulink Blocks [49] 62
5.2 SIMULINK Block Equations [66] 63
5.3 Supported Simulink Blocks 69
5.4 SIMULINK to MRICDF Type Mapping 72
5.5 Overclock Trace (K = 2) 73
5.6 Underclock Trace (K = 3) 74

Chapter 1

Introduction

Embedded systems are nearly ubiquitous in modern society. From fly-by-wire and drive-by-wire to the control of power plants and medical equipment such as pacemakers, the amount of control software driving safety critical tasks has been steadily on the rise. The complexity of the systems they operate has also increased over the years from simple sequential processors to multi-core parallel processors operating in a distributed environment. Developers therefore face an increasingly difficult, and seemingly counter-intuitive, challenge of creating code for such systems that is both safe and reliable while also minimising the cost and time to market in order to keep up with commercial demands. These distributed real-time embedded (DRE) systems must therefore place the correctness of the product as their highest priority, yet the productivity can't be ignored. Model driven engineering (MDE) is a common approach for easing the burdens of verification by providing a specification at a higher level of abstraction and automating the engineering process as far as possible. Manual programming of concurrency is tedious and error prone, which MBE can alleviate by providing code generation facilities. For the purposes of safety critical systems its preferable to generate this code automatically from a formal specification - one with a mathematical basis and rigorously defined semantics suited to formal analysis. More reliable code generation is only one aspect of the problem, too often a system is validated at a high-level of abstraction only to discover that the actual target platform or environment render the deliverable generated from the specification infeasible. For this reason it is advisable to employ a model that captures not only the system's software hierarchy and behavior but also the properties of the target platform to enable early detection and verification, thereby minimizing time between specification and error detection and subsequently the cost to correct discovered errors.

A number of MBE standards have been put forth and adopted by various industries. The Object Management Group (OMG) [6] has produced many modeling languages [28, 31, 61] as part of their Model Driven Architecture (MDA) approach to MBE. The MDA approach divides the development process into three distinct phases. The Computation Independent Model (CIM) describes the requirements of the system from an end-user perspective without

necessarily going into the details regarding the intended structure or implementation. As such it typically is presented in natural language at the business or domain level. From the CIM a system architect creates a Platform Independent Model (PIM) that models the structure and composition of the system at a level that abstracts away the technical details of the underlying platform. In this way a PIM can be mapped to multiple potential target platforms. Finally, a specialist in the chosen platform creates a platform specific model (PSM) combining the PIM specification with a platform model (PM). While the PIM is given at a higher level of abstraction using a language like UML[61] the PM is created using an appropriate lower level standard such as CORBA[4] or Microsoft's .NET [5]. The PSM then represents the final refinement of the original CIM specification from which an executable or other appropriate deliverable may be generated.

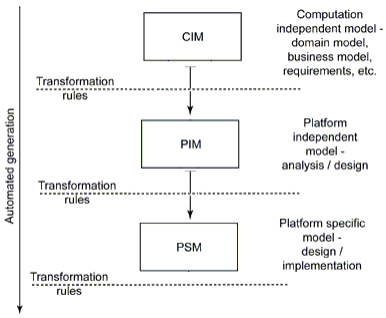

Figure 1.1: Model Driven Architecture Approach

In addition to general solutions such as MDA, some industries have created their own domain specific approaches. The automotive industry, for instance, has the AUTomotive Open System ARchitecture (AUTOSAR)[3]. AUTOSAR is an open standard defined by a consortium of manufacturers in the industry. Its purpose is to provide a standardized methodology and interface for the development of automotive systems. To accomplish this, AUTOSAR defines a multilayered abstraction between the vehicle's microprocessors and any application software. This abstraction has a well defined, standardized interface so that changes can be made within the application or hardware layer without disrupting each other. Another industry example can be found in the avionics industry. The *Integrated Modular Avionics* (IMA) [20] approach popularly employed in modern avionics design, allows multiple applications to operate on a shared computation platform. Each application is partitioned into its own encapsulated execution environment with memory space reserved for it during system configuration. *ARINC 653* [9] is an industry standard for the specification of an APplication EXecutive (APEX). The goal of the APEX is to provide a layer of abstraction between the application partitions and the underlying shared computational platform. This provides portability and reuse for applications due to ARINC 653's standardized interface, and improves hardware modularity by insulating the application layer from changes to those components.

1.1 Problem Statement

Despite the advantages provided by these existing MBE tools and approaches in terms of productivity and savings, a number of problems still remain. One major concern is how to validate the consistency between different abstractions. Consider OMG's approach, each layer is conceivably created using a different abstraction and must be translated at each transition between development phases. This fact alone increases the amount of effort that must go into each design but it is compounded by the fact that if an error is discovered that requires correction especially at a later stage of development it can cause a cascade of corrections stemming from the source of the error in a previous phase. AUTOSAR's multilayered abstraction between the software applications and the hardware provides code reuse and modularity. However, to integrate user code, AUTOSAR generates code skeletons that must be manually completed by the developer. These aren't updated automatically and therefore must be manually edited, or new skeletons added, after each system revision. There is no guarantee that the original semantics of the user code is preserved during the code generation process [64]. A common problem in these standards is the lack of a formal semantics necessary to enact correct-by-construction translations for the refinement of a model to a lower abstraction or for the generation of executable code. Manual implementations and translations introduce a greater potential for error and require additional testing to verify that the implementations are consistent with the source specifications.

Addressing these issues requires a methodology that uses a single modeling formalism for its end-to-end development. The model would need to encapsulate the software architecture, behavior, and the underlying execution platform. Further, the modeling standard must be extensible to support the addition of (non-)functional properties (power requirements, scheduling, etc.) as needed by development tools to facilitate analysis. A single model based methodology can be rapidly revised and reanalyzed. Introducing hardware to the model at the same phase as the software, behavior, and requirement properties allows the user to detect performance and resource inadequacies earlier in development. Ease of revision and early verification allows for thorough optimization and testing with an "evolutionary" approach [43] to prototyping. Finally, specifying component behavior with formal semantics allows for correct-by-construction synthesis of the final code. The Architecture Analysis and Design Language (AADL) [63] is an architecture description language (ADL) that was originally developed to create a comprehensive model of a systems architectural hierarchy - including both the software and the underlying execution platform. AADL is an extensible standard with an inbuilt capability for user defined properties and units of measure. Furthermore, it permits the definition of language annexes when it is necessary to extend the core semantics for the modeling and analysis of user defined properties. The Open Source AADL Tool Environment (OSATE) is the de facto editor for AADL, providing syntactic and semantic analysis of textual specifications. Optionally, the developer may make use of available plug-ins for a graphical front end and some static analysis methods such as real-time schedulability analysis of the thread components [65]. What OSATE doesn't currently do is

provide a means of meaningfully utilizing the code associated with the software components. Code may be referred to by a property that specifies the name and file path as strings, but such files are not accessed by OSATE. Alternatively, developers may use the Behavioral [62] and Error Model [29] Annexes to manually specify and simulate the intended operational and error state behavior of a component, but this approach requires manual maintenance and introduces the potential for conflict between the modeled behavior and that of the actual source code. Ocarina [36] is another third party tool that was introduced to provide code generation support for AADL. Ocarina is capable of syntactic and semantic parsing of textual AADL specifications. In addition, by accessing the source code files referenced by the properties of software components in the model it can generate the distributed code that will execute atop a middleware on the underlying computational platform. Currently, Ocarina is configured to accept the general programming languages Ada and C/C++. More recently it has experimented with the limited inclusion of the synchronous languages Esterel and Lustre. Therefore we wish to determine whether an End to End approach to model driven engineering using polychronous behavior specifications is feasible using a single modeling formalism throughout. We believe that an approach rooted in AADL's comprehensive, extensible standard and supplemented by Ocarina's automated code generation capabilities would address the aforementioned issues of delayed hardware integration and throwaway prototyping. However, there is still the issue of the informal semantics and behavior validation. While the *syntax* of these languages are typically well defined this only describes the structure of a well formed *sentence* in the language. The actual meaning of these sentences, their *semantics*, are understood through descriptions in natural language and examples with the developer's intuition being relied upon to fill in any gaps in understanding. Without well defined, mathematically grounded semantics to provide a means of reasoning about the correctness formally, one must rely on costly exhaustive testing techniques which are time consuming. Moreover, such techniques can miss errors as they cannot be guaranteed to be able to explore all possible execution states. The issue of error detection is compounded by the fact that the ambiguity of the original semantics itself can cause developers to inadvertently create errors while programming.

In this dissertation we propose a methodology and present a corresponding prototype that implements that approach. Our goal is to design safety critical systems such that the final product has a minimal incidence of defects or behavioral bugs. In order to achieve this goal we hypothesized that a unified formal system specification that could be refined through equivalence preserving transformations would produce verifiable, implementable models for critical embedded systems. Such a process would reduce the time to market and the costs of post production validation with its modularity, code reuse, and formal verification throughout development. To test this hypothesis we defined a methodology for the end to end development of safety critical systems. We then implemented a prototype framework that incorporates multiple existing tools and languages as well as extensions and newly customized tools. This new framework is then used to create example systems. The example case studies are used to evaluate the framework by comparing the generated results to the original specification. We look for *flow equivalence* between them. By flow equivalence we mean

we wish to determine whether the variables that belong to both the specification and the implementation have the same behavior trace for the same input stimuli.

Our methodology is based on the previously described AADL standard. We utilize a code generation process based on the work described in Ocarina[36] to generate distributed code that runs on a high-integrity middleware. We've extended the code generation process to support polychronous specifications provided in MRICDF[41]. Esterel [23] and Lustre[33] based specifications have been attempted. These languages rely on synchronizing with a single global clock. On the other hand, polychronous semantics allow different components to operate at independent rates. We believe this will be advantageous in modeling and implementing a distributed system. A polychronous approach frees the components in a distributed system model from the need to synchronize with each other unnecessarily. In addition to providing rigorous formal semantics for validation, we can also leverage MRICDF's epoch analysis to automate the implementation and validation of concurrent threads from a process level specification thereby mitigating the cost and risk that would be present with manually created parallelism. Although correctness is of paramount importance, we also recognize the value of usability and productivity. To that end we believe that its worthwhile to support the use of other modeling formalisms as a frontend for the initial behavioral specification. By translating that model into a formal intermediate format we can allow developers to benefit from existing tools and experience while still leveraging the analysis and code generation capabilities of our polychronous model of computation. For our prototype implementation we have chosen to implement a Simulink to MRICDF translation. This translation process is a continuation of the work started in [49] wherein Simulink was translated to the polychronous language Signal[46]. We've chosen Simulink for the fact that it is a widely known and supported modeling language, having become a de facto standard in many application domains. We've created an automated translation that extracts and validates the typing and timing properties of the original Simulink model and then applies that information to an equivalent MRICDF actor network created by a hierarchical, top-down translation of the Simulink blocks. this is not to say that this approach or the prototype framework are limited to only implementing models from Simulink specifications. While we chose these languages for our prototype, the methodology is applicable to and combination of ADL modeling language with a polychronous behavioral specification. The framework is flexible enough to support the future addition of other frontend languages or for the generation of source files in languages other than C/C++.

1.2 Contributions

The contributions of this dissertation are as follows:

Methodology: We describe a new methodology for evolutionary prototyping, verification, and embedded system development. Through the use of ADLs and polychronous software, systems can be rapidly prototyped and iteratively refined until they are ready to be imple-

mented by automated generation and distribution. The novelty of this approach is in the extent to which we have integrated the use of the ADL model and Polychronous behavioral specification for the purposes of end to end development. By providing front end support for the existing modeling standard and the translation tools needed to convert them to a formal intermediate format we aim to automate support for legacy and existing industry standards.

Tool Suite: We have been developing a tool suite to support this methodology. The tools are based on Ocarina, but with extensions to incorporate polychronous specifications given in MRICDF. To the best of our knowledge this is the first time a polychronous language has been used directly as a behavioral annex for AADL. An additional phase has been added to the mode analysis and code generation step of Ocarina while handling MRICDF behavioral specifications, so that MRICDF associated processes can be analyzed and have a multithreaded implementation for them generated and the AADL model can then be updated accordingly. We also built an automated translation tool to adapt Simulink models to an intermediate representation in MRICDF for formal verification and code generation. In doing so we also explore and innovate on relations between polychrony and Simulink.

Case Studies: We provide two case studies to illustrate the core aspects of our methodology. The first is an elevator implementation that demonstrates the APECS process for modeling a safety critical system. The scope of this example begins with modeling the underlying platform and software hierarchy and progresses to the generated the executable code. Our next case study is a phasor measurement unit (PMU). This PMU has been manually implemented in both Simulink and MRICDF to provide a comparison between a native MRICDF implementation and the result of a Simulink translation.

1.3 Overview

The remainder of this document is organized as follows:

Chapter 2 describes the state of the art for Model Based Engineering. We examine the UML family of languages and its application in industry tools. We also look at the MATLAB / Simulink language and the Architecture Analysis and Design language that feature strongly in our own proposed methodology.

Chapter 3 covers the preliminary details of formal languages and correct-by-construction code synthesis. We discuss the goals and general philosophy of the synchronous languages and give an overview of the most well known formalisms and a comparative look at their general approaches.

Chapter 4 provides the details of the proposed APECS methodology. We discuss the component tools such as Ocarina, OSATE, and Cheddar that form the core of the proposed development environment as well as descriptions of the roles of the languages and standards such as AADL (for providing the system model) and MRICDF (polychronous software spec-

ification) that provide the source models for the extended tool chain.

Chapter 5 presents the Simulink frontend extension for APECS and the translation process from Simulink to an MRICDF intermediate format for verification. Simulink models are attached by property definition to process components in the AADL system model. They undergo type and clock extraction and validation before being hierarchically translated into a behaviorally equivalent MRICDF model.

Chapter 6 contains a number of case studies that demonstrate the methodology, translation, and code generation processes of APECS. We present the design of an Elevator Control System in APECS, generating code for the safe operation of the car doors. We use the design of a Phasor Measurement Unit in both MRICDF and Simulink to compare and contrast the translation capabilities of our Simulink front end.

Chapter 7 describes our conclusions and the potential future work for this project.

Chapter 2

Model Driven Engineering

Low-level languages have long been the choice for the development of embedded software. The reason for this choice was their closeness to the underlying hardware platforms which allowed for detail oriented approaches that could minimize the memory footprint of the program. Over the years the capabilities of the hardware increased along with the complexity and utilization of embedded systems in safety critical applications. This led to a gradual evolution in methodologies that shifted the focus from minimizing hardware requirements to reducing the development time and the cost of verifying the systems. Model driven engineering (MDE) is one solution that has been applied to this end, reducing costs and maximizing the viability of the software. MDE allows developers to create specifications with interactions between different languages and to build libraries of dedicated models for reuse in domain specific tasks. These models allow critical behaviors to be predicted before implementation and, through transformative techniques, for the automated generation of a final product.

The idea to simplify complex systems for study by creating an abstraction is an old one. In computer science applications, however, the order of creation is inverted. Rather than using the model to observe and better understand an existing system, the model is created in an earlier stage to be used to anticipate the behavior of a future system prior to its implementation. These models capture important aspects of the system being studied while abstracting away details yet to be determined in future design decisions or those that are considered unimportant to model validating concerns. They describe the intended behavior or structure of the desired system and one or more potential configurations that may achieve the intended results. This allows the model to be more specialized towards specific forms of analysis than the final system would be. Capturing these requirements and domain specific properties in the model is important to allow stakeholders (those with an interest in the product being modeled) of the system under design to conceptualize and agree upon the requirements and initial specification. A *specification* describes what the target does while the *implementation* describes how it is done. Ideally the specification should be completed

first because an implementation created from an incomplete specification runs a significantly greater risk of failing to fulfill its design requirements. Indeed MDE as a whole allows the developers to visualize the design and its possibilities. The ease with which the model can be changed relative to a low-level language implementation is not the only factor in MDE's versatility either. An abstract model allows the developer to organize and edit information about a large, complex system. Variations of this system can be saved separately, allowing teams to explore the relative virtues of multiple potential solutions without necessarily needing to implement them first. There are multiple levels of abstraction that may be employed in MDE. The highest level models serve a guiding role, being built very early in the design process. These models focus on the key concepts of the intended system's purpose and features. The goal of these guiding models is to capture the system's requirements and highlight the design options available to the stakeholders before the implementation is too far along. The majority of details regarding actual implementation are abstracted away to focus first on establishing and correcting the high-level concepts. A Computation Independent Model employed in the first stage of UML model based analysis approach is an example of this. As the design process proceeds, the model is refined, additional implementation details are added and the model grows into an abstract specification of the system's structure, behavior, or both. By this stage, key concepts and components should have their mechanisms modeled. This level of abstraction will likely have some amount of correspondence to the final product. Although some details are yet to be implemented, they will eventually be described as the model evolves towards completion. Eventually a full specification will be obtained, containing enough information to build the final system or to generate it through a model transformation. This is a still a distinct case from prototypes, or examples, which are created using specific instances. While a model describes its subject, an instance is most typically part of an actual run-time executable. As such, while the final specification is still represented using the same semantics and tools as previous general case models, albeit with greater detail, the instances are specific examples. There are a few notable issues with this approach to which MDE practitioners usually pay attention. First, there must be traceability between the iterations of the model as it is refined. There is a definite distinction between refining a specification as design decisions are made and the more random approach of trying new implementations until a solution is found. Second, there is the issue of abstraction. Determining which mechanisms, properties, and resources are *essential* to the model and which can safely be abstracted is ultimately a judgement call on the part of the developer. The details needed for code generation are not necessarily the same as those for a simple cost analysis. The analysis of those attributes deemed worth modeling allow for the deduction of the final system's properties. Through techniques such as co-simulation and verification, MDE endeavors to ensure the ultimate product is a trusted, high performance software. As a result, care must be taken when abstracting elements of the system. If done poorly the model may cease to accurately represent the intended system. For instance, if a component were erroneously changed from a discrete to a continuous type the fundamental behavior of that aspect of the system would be altered. There is no clear dichotomy to choose between a *detailed* and an *abstract* model. Rather there is most commonly a spectrum of

precision that a model may progress through as it develops over the life of the project. In addition to the level and choice of , there is also the matter of the context of the model to consider. The model for a system may be either *open* or *closed*. A *closed* system is self contained, it models both the intended system and the environmental elements with which it interacts. An *open* system on the other hand simply models the intended system and treats its environment as a black box. Thus it can be said that the component subsystems of a closed system can be considered open systems. This is especially true of models of open systems that are encapsulated in a closed model representing a testing environment based on development requirements and assumptions. Also, while its not semantically required, a large and monolithic model cannot be effectively distributed among development teams working concurrently. As a result, most teams in such a distributed development process will be working with these open subsystems. Annotations are employed by models to track assumptions about the environment in these situations as well as design decisions and versions by the development teams on the project. The internal structure and organization of these models will be made up of structural and behavioral models. Both are important, the structure of the model is needed to determine and allocate the resources of the system while the behavior model is necessary for the eventual generation of the embedded software. While the latter is required to identify problematic or emergent system behaviors, this analysis typically assumes that a model of the intended system structure exists. Often these models will be created using different semantics or (subsets) of languages. For instance, the structure of a UML model will be created using class diagrams while the behavior would be created using State Charts or Finite State Automata. The MDE approach can be very beneficial in terms of the reduction of the cost and time required to deliver embedded software. It can greatly simplify the system for analysis and the verification of safety critical properties. However, it is also a challenging prospect. The model must be created carefully and well documented for the traceability necessary to verify that any subsequent refinement and transformation conform to the original specification. Indeed, the final result will only be as valid as the guiding specification model whose assumptions and abstractions must be chosen with care. The models are not themselves programming languages nor do they inherently possess of a methodology for creating the models they describe. Next we will look at some of the state of the art languages and tools for MDE, how they are utilized, and how they approach these issues.

2.1 UML

The unified modeling language (UML) is a general purpose modeling language for the specification, visualization, and documentation of software systems. It was first introduced in 1994 as a unification three approaches to object-oriented design: the Object Modelling Technique (OMT) [60], the Booch method [21], and the Objectory method [39]. Modeling is about abstracting the object of study to allow the developers to focus on particular properties or features for study. Rather than creating a singular syntax for representing all the possible

areas of study and analysis, UML provides a framework that supports numerous design diagrams each of which specializes in representing different aspects of the system being studied. Figure 2.1 presents the 14 diagrams that belong to the UML 2.0 standard. These diagrams can be broadly divided into two groups, one describing aspects of system behavior and the other detailing the structure of the system.

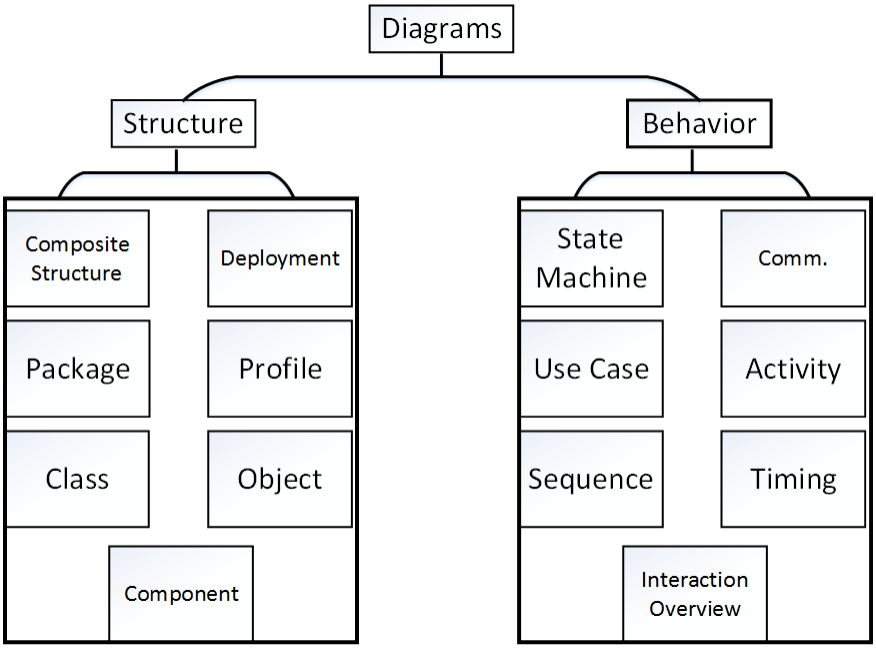

Figure 2.1: UML Diagrams [61]

The structure diagrams provide the facilities to conceptualize the hierarchy and resources needed to implement the system.

- The *Class* Diagram is the cornerstone of UML's structural modeling for object oriented design. The classes contain the operations and attributes that each are or should be capable of as well as indicating the relations between classes. Each class is divided into three fields: name, attributes (variables), and operations (functions/methods). The relationships are encoded as different types of connecting arrows, potentially annotated with additional information.

- The *Component* Diagram represents the structure of components within the UML model. These components could be logical (business, organizational, etc) or physical. The component diagram defines the relationships between these components through establishing interface requirements: defining ports and connections.

- The *Composite Structure* Diagram describes the internal composition of a structured class. Conceptually it is similar to a component diagram except it graphically represents the decomposition of a class into its constituent properties and parts. Thus data

and control flows through the class as well as the relationships between parts can be visually diagrammed.

- The *Deployment* Diagram shows the relationship between the software architecture of the system and physical targets. Depending on the implementation chosen the deployment may describe the architecture at the *specification* level or the *instance* level, with the former not referencing any specific instantiations within the hardware platforms or software classes.

- The *Package* Diagram, unsurprisingly, showcases the dependencies between packages. A *Package* in UML functions much as a library file does in C/C++, serving to hierarchically collect other diagram and even other packages under its namespace. The package diagram shows the contents of a package and how these contents interact with each other.

- The *Profile* Diagram is the mechanism by which custom additions may be made to UML. These additions take the form of *stereotypes*, *tagged values*, and *constraints*. It is somewhat limited in that it does not allow for fundamental changes to the UML metamodel but rather for the customization to better fit a specific domain.

- The *Object* Diagram, though obsolete after the UML 2.4 specification, described a particular set of instances of the system. It closely resembles a class diagram in usage and notation, but while the class diagram showed generalized version of the classes, the object diagram showed specific instances of these classes. The diagram was therefore a more concrete representation suited for prototypes or test cases of a given class diagram.

The behavior diagrams allow the clients to specify their expectations in terms of the intended user base, system capabilities, and desired reactions to particular situations. It's also intended to allow developers the capability to visually communicate the structure and interactions of the blocks.

- The *Use Case* Diagram, also referred to as the *Behavior Diagram* is used to describe the intended external users of the system, the key components of the system itself, and how the two should interface with each other.

- The *Activity* Diagram, on the other hand, describes the behavior of a component in terms of a flowchart or *workflow*. It is sometimes used as a simpler alternative to the State Machine diagram.

- The *State Machine* Diagram serves a similar purpose to the activity diagram, describing the behavior of a component. However, as the name suggests, it employs a State Machine to describe discrete behavior in terms of states that are connected by transitions triggered by the arrival of specific events.

- The *Sequence* Diagram describes inter-component behavior rather than the intra-component descriptions found in the activity and state machine diagrams. It shows the interaction between objects and the order in which these interactions transpire. They're designed to show the sequence of interactions for a particular scenario so that the businesses could detail the current or desired behavior in specific circumstances.

- The *Timing* Diagram is similar to the sequence diagram in that it describes the behavior of a component in a particular time frame. It may be used to describe individual components or it may be used with multiple components to describe their interactions in that span of time.

- The *Interaction* Diagram is a variant of the activity diagram that specializes in displaying the control flow of the modeled system components. Indeed it may inline other interaction models (sequence, communication, timing, etc) within *frames* inside the interaction diagram to detail other forms of interaction that occur when the control flow reaches that point.

- The *Communication* Diagram, previously referred to as a collaboration diagram, is similar in purpose to a sequence diagram but instead focuses on the messages being passed between objects. Indeed, often the same information can be communicated with either diagram but the order of the messages is more readily apparent in the sequence diagram while the communication diagram more explicitly shows the full set of messages.

While it currently enjoys significant use in both system engineering and business requirement specification, this high adoption rate exists despite the fact that UML's development process produces diagrams as separate entities and then relies on the designers to maintain consistency across the models. While UML does focus on object-oriented development it is not a programming language nor is it a methodology in itself. Instead it presents a standard containing a set of complementary diagrams with loosely defined semantics for use in building other methodologies and tool suites. These diagrams form the foundation of UML but the language can be extended in a few ways to support other modeling domains and analysis techniques. *Stereotypes* define new model elements derived from existing elements, allowing the user to customize the component for a specific domain. The stereotypes may have new special attributes and graphical representations to tailor themselves to their intended role. *Tagged Values* are keyword / value pairs that allow the designer to define properties of the class being modeled, while *constraints* are model invariants that are expressed formally and explicitly using the Object Constraint Language (OCL). Yet these extensions proved insufficient when attempting to apply UML to the software engineering of real-time embedded systems, and so related specialized languages were created to approach these areas of design.

2.1.1 SysML

UML's adoption by industry in software development along with tool development efforts led the systems engineering community to consider adopting it for systems modeling. In 2003 the International Council on Systems Engineering (INCOSE) declared it the common language for systems engineering. However, while its object oriented focus is well suited for software development, it suffers from a number of limitations at the system modeling level. Thus in the 2005 release of UML 2.0, OMG attempted to introduce new concepts and language extensions to improve its applicability for systems modeling. While several important diagrams such as activity, sequence, and composite structures were added, more improvements were needed. Collaboration between OMG and INCOSE to further extend and adapt UML 2.0 for system modeling evolved into a new language called SysML.

SysML is not just a specialized subset of UML. While it shares some diagrams with UML and modifies others, it also adds a whole new class of modeling diagrams and semantics for specifying requirements and constraints. Figure 2.2 shows the diagrams available in the SysML standard.

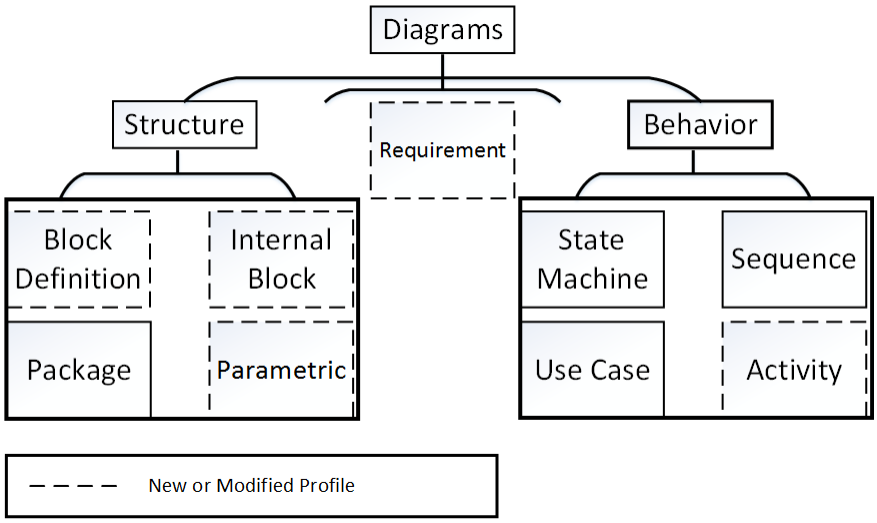

Figure 2.2: SysML Diagrams [31]

Many of these diagrams are available in UML, but some have been modified. Here we will briefly highlight the modifications and features added by SysML to those diagrams that were retained.

- The *Activity* Diagram is very similar to that of UML. It describes the control flow between actions and along conditional branches for the component(s) being described. SysML, however, extends the UML 2.0 functionality, through the use of stereotypes, with the ability to describe the rate of the flow - whether continuous or discrete. This

enables the modeling of continuous systems with SysML activity diagrams, although by default it is assumed to have a discrete behavior.

- The *Block Definition* Diagram contains block definitions and models their relationships with each other. In SysML classes and components are replaced with the more neutral concept of a *block*. The block definition diagram therefore serves to replace the class diagrams of UML. See Figure 2.3 for a brief example of a SysML block.

- The *Internal Block* Diagram (IBD) describes the internal structure of a block, taking the place of the composite structure diagram from UML 2.0. Like the composite structure diagram, this IBD allows for multiple layers of nested composition. One addition made by SysML to this diagram is the support for flow ports and connections between such ports on different blocks, describing the movement of the flows within a block's structure as well as between blocks.

Additionally, a pair of entirely new diagrams were added. The focus of these diagrams is on providing the means for the analysis and traceability of requirements and properties of the system being modeled.

- The *Requirement* Diagram can be created as a package containing a hierarchical representation of one more more requirements and their relationships to each other. SysML defines four such possible relationships that indicate how each requirement *refines*, *derives* from, or is needed to *satisfy* or *verify* that the other requirement is met. In addition to packages, a requirement block may be defined directly within block definition and use case diagrams. In these cases the requirement serves to graphically encapsulate a textual specification of a requirement of the block or use case with which its associated. This category of diagrams also allows two types of graphical notes that can be added to annotate any type of block or diagram: *problem* and *rationale*. Problem notes describe a problem to be solved while the rationales provide a justification. For an example of a requirement block refer to figure 2.4.

- The *Parametric* Diagram describe constraints on the parameters of the model. SysML does not provide a specific constraint language, allowing the user to specify their constraints formally (MathML/OCL) or informally. These diagrams merely serve as a means to record and visually present information on parameter constraints, SysML itself does not provide analysis tools to determine whether they have been satisfied.

While these diagrams partially overlap with those used by UML, the modifications and new additions make SysML better suited toward Systems Engineering than its UML progenitor. These changes were necessary to overcome practical and psychological hurdles faced by system engineers working with UML. For instance, the concepts of *class* and *objects* that were so well matched with the object oriented design of software are replaced with the more generic concept of a *block*. Blocks are the basic modeling element of SysML, depending on how they

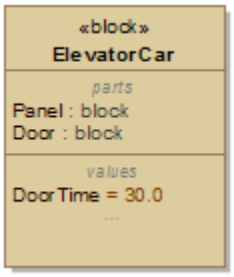

Figure 2.3: Elevator Car Block

are defined they can model physical entities of the model platform or model logical/conceptual entities. Blocks are defined in the *block definition diagram* which graphically represents blocks as compartmentalized rectangles. Each compartment specifies different properties, although only the *name* compartment at the top of the block is required to form a valid definition. Additional properties that may be specified in order to describe the decomposition of the blocks or to provide the values of quantifiable characteristics of the block. Each compartment is also labeled to indicate the type of the properties it contains. For example, the block show in 2.3 shows a block that encapsulates the structure and behavior of an elevator car. This block contains two sub-blocks: Panel and Door. These blocks respectively represent the button panel interface for the elevator and the sensors and apparatus that control the opening and closing of the door. Finally it contains a value representing the desired duration a door should be held open before automatically closing.

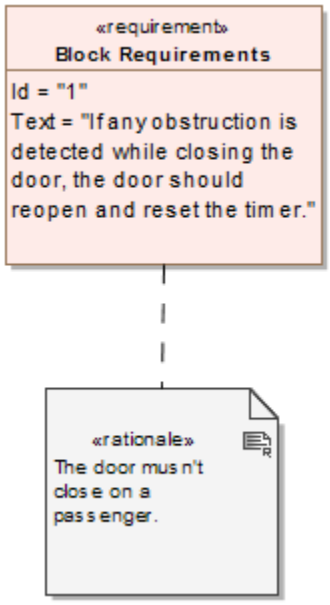

Figure 2.4: Elevator Car Requirements

While UML 2.0 has diagrams for representing the structure and behavior of the target sys-

tem, SysML adds a third category of diagrams for the textual specification of requirements and for constraints on block properties. In figure 2.4, we see a sample requirement block for the elevator car. Requirement blocks each have a unique numeric identifier as well as the specification text. This requirement block describes in plain text a requirement that the door not be shut while an obstruction is present. While SysML does not provide tools or methodologies for verifying that these specifications are satisfied, and indeed a natural language specification would not be well suited to this task, it is useful for providing traceability for developers refining the block over the life of the model's development. Additionally, we see the specialized comment providing the *rationale* for the presence of the requirement, in which stakeholders can clarify the motivations for setting the requirements as they have.

2.1.2 MARTE

While SysML added many necessary features for System's Engineering, it still had notable deficiencies when applied to the development of Real-Time Embedded Systems. Considering, for example, the implementation of a time-triggered, periodic program running on a multicore system. At first glance this would appear a simple prospect for implementation in UML. The software application can be modeled as an active class along with the external clock and devices. The time between activations can be given as an annotation of the activity diagram inside the application class. However, were we to attempt to generate code from such a model, UML would be unable to distinguish between the classes that should become software and those that are devices in the environment. Additionally, the property annotations are strictly numeric values with no means of explicitly declaring whether the time is in seconds, milliseconds, etc. While one could assume a base unit of measure for use in the model this can quickly become cumbersome at an industrial level and is highly error prone. Worse yet, with no way to distinguish the software and platform components or create explicit allocation bindings between them, its impossible to test whether the desired real-time parameters of the system will be fulfilled from the model. Thus in 2009, after much collaboration between researchers and manufacturers, the first version of the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) norm was published. Like the UML 2.0 standard on which it is based, MARTE is not a programming language or a methodology in itself to be used directly. As with its predecessors, it is a collection of specialized packages meant to be modular in application and is extensible. It can be said that there are primarily two users of the MARTE norm. The first are *methodologists* that will define the modeling capabilities required for specific domains and then create the language extensions necessary to fulfill these requirements. The *producers* take the results of the work done by the methodologists and implement the tools, and techniques necessary to put it to use in their field. We will not attempt to discuss the entirety of the MARTE norm here as it is an extensive document at over 800 pages. Instead we will provide an overview of the packages provided by the norm and highlight their usefulness for real-time systems.

The foundations of MARTE provide the core concepts and profiles needed to define the

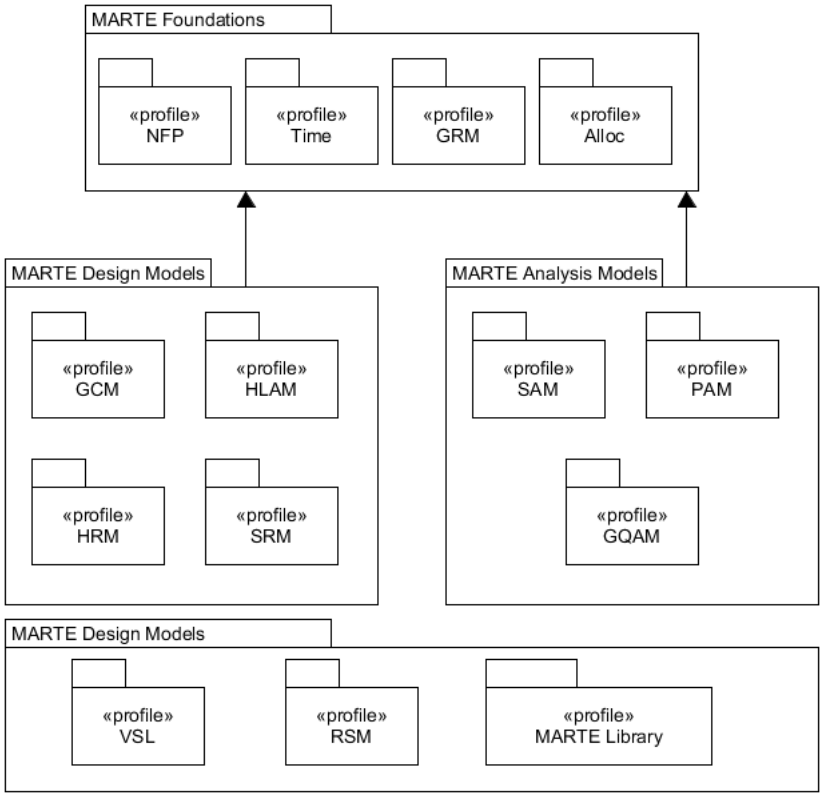

Figure 2.5: MARTE Architecture [28]

operational modes of a system.

- The *Non-Functional Properties*(NFP) in conjunction with the Value Specification (VSL) declares, qualifies, and applies non-function property information to UML models.
- The *Time* profile proposes multiple distinct models of time for embedded real-time systems. The first two are the standard chronometric model of time and one based on a synchronous paradigm. The final temporal model is the Clock-Constraint Specification Language (CCSL) that complements the primary Time profile by allowing the users to describe constraints between the clocks.
- The *Generic Resource Modeling*(GRM)forms the basis of MARTE's platform modeling capabilities. This is at a higher level of abstraction than the norm, modeling the resources at a system level rather than hardware or software elements which are handled into more specialized sub-profiles (SRM and HRM).
- *Allocation* enables the explicit mapping of software applications to elements of the underlying hardware platform.

The *Design Model* provides the profiles that specialize in the project definition and design aspects of the model based development process.

- The *Generic Component Model*(GCM) is essential for the specification of the core architecture of embedded systems. It extends UML's composite structures with domain-specific features such as the capability to associate behavioral and structural elements of the model.
- *Software Resource Modeling*(SRM) provides the core elements for modeling the software components of the system.
- *Hardware Resource Modeling*(HRM) similarly provides the capability for modeling hardware platforms.
- The *High-Level Application Modeling*(HLAM) profile provides annotations for specifying the real-time and embedded properties.

The *Analysis Model* profile library contains *Generic Quantitative Analysis Modeling*(GQAM) which provides the features for model-based analysis and is supported by two refined sub-profiles *Schedulability Analysis Modeling*(SAM) and *Performance Analysis Modeling*(PAM). These profiles extend the modeling standard with a number of key features for real-time embedded systems modeling and analysis not present in the core UML standard or in the SysML stand-alone variant. The added capability to specify qualitative and quantitative measures using explicit units, implement component timings using various temporal models, and distinguish between hardware and software components are invaluable improvements over the plain UML norm.

2.2 MATLAB/Simulink

Simulink is a simulation-based, model driven design tool using block diagrams to describe dynamic systems. Simulink models are organized as functional blocks representing piecewise-constant functions that operate on discrete-time signals. The primitive blocks of the model can be broadly grouped into two categories: Interface and Combinatorial. Interface blocks connect the model with its environment and fall in to the subcategory of sources and sinks. These connections may be to an external source (e.g FromFile, FromWorkspace), to another Simulink system (Inport, Outport), or to something internal (Constant). Combinatorial Blocks are those that modify the values of the signals as they move from source to sink, applying operations to one or more inputs and instantaneously producing one or more outputs. These operations may be arithmetic, relational, logical or something more advanced. Subsystems are user-defined abstractions that encapsulate primitives and other subsystem blocks. They define an interface of input and output data ports that creates a standardized

interface between the environment and the system's internal components. The internal components of default subsystems may inherit their sampling rate from their inputs or they may have their own explicitly defined timings. There also exist two variants of the system with differing implementations of control for the system's execution. The triggered subsystems receive an additional control signal. The system and its subcomponents only activate during instants when a triggering event occurs. A Triggering event is reflected by a change in value of the control signal either rising, falling or both depending on the user selected settings. Alternatively, an enabled subsystem operates during instants where the control input, 'enable', is set. Both of these subsystem variants affect their subcomponents beyond merely establishing hierarchy, complicating the timing requirements in ways that will be explored later in Chapter 5.

2.3 Related Work - Formalizing Simulink

2.3.1 Formalizing SIMULINK's Behavioral Semantics

In [22] the authors propose defining formal semantics for the SIMULINK simulation engine. The state-space of SIMULINK's dynamical system is represented as a system of equations: a *continuous-time state function* $f_x : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \rightarrow \mathbb{R}^{n_x}$, *discrete-time state function* $f_d : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \rightarrow \mathbb{R}^{n_d}$, and an *output function* $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \rightarrow \mathbb{R}^{m}$. x is the continuous state and n_x is the number of continuous state variables. d is the discrete state and n_d is the number of discrete state variables. m is the number of outputs. The system of BNF form equations were generated automatically by inspecting the *mdl* source file of the model. Currently only a limited number of blocks are supported (Seen in table. 2.1). The chosen blocks come from both the continuous and discrete libraries. A closed

Table 2.1: Supported Blocks [22]

Library	Blocks	Equations
Source	Constant	$\ell_1 = constant$
Sink	Output	out1 = $out1 = \ell_1$
Arithmetic operations	Add/Mul	$\ell_3 = \ell_1 + \ell_2$ or $\ell_3 = \ell_1 \times \ell_2$
Signal Routing	Switch	$\ell_4 = if(p_r(\ell_2), \ell_1, \ell_3)$
Continuous-time	Integrator	$\ell_2 = x; \dot{x} = \ell_1; x(0) = init$
Discrete-time	Unit Delay	$\ell_2 = d; \bar{d} = \ell_1; d(0) = init;$

form solution for the discrete and continuous equations is generally not possible, instead the solution is approximated using a small step size temporal discretization and a numerical

solver that is accurate over small time intervals. While this approach formalizes the semantics of SIMULINK numerical solver, it is limited to a small subset of blocks. Also, unlike the translation solutions, there are no known existing tools to exploit these new semantics for static analysis.

2.3.2 Hybrid-Automata based intermediate formats

Hybrid-Automata [35] are finite state machines that are used for modelling systems that include discrete and continuous interaction. It models both the discrete behavior of the software and hardware and the continuous behavior of physical, dynamic components. A hybrid automata is a tuple of the form (S, s_0, V, P, T) where

- S is a set of states (also referred to as *discrete locations*)
- s_0 is the initial state, $s_0 \epsilon S$
- V is a finite set of typed variables
- P is a set of parameters
- T is a set of transitions

In [11] the authors transform Matlab Simulink/Stateflow (MSS) models into an intermediate format called the Hybrid System Interchange Format (HSIF) [55]. HSIF describes the MSS model as a network of hybrid automata described by the tuple (HA, V, P, C) where

- HA is a Hybrid Automata
- V is a finite set of variables, which are partitioned into signals(input and output), local variables V_l, and shared variables V_s
- P is a set of parameters
- C is an input constraint

HSIF was created with the goal of *semantic interoperability*, allowing an interchange of models between different tools for simulation, verification, and code synthesis purposes. Each node or *location* contains a system of equations: differential and algebraic. The differential equations describe the continuous time behavior of the system while the algebraic functions represent dependencies between variables. The authors of [35] have implemented a tool they call Graph Rewriting and Transformation (GReAT) [10, 42]. GReAT translates the MSS model by the following steps:

Step 1: **Enumerate switching signals** - these output signals drive switch blocks that can change the behavioral structure of the system, therefore they must be identified and tracked.

Step 2: **Transform MSS states to HSIF locations** - determined by analyzing the MSS model's Stateflow machine, combined with the identified switching signals.

Step 3: **Transform transitions** - map the Stateflow transitions to HSIF location transitions.

Step 4: **Generate Behavioral Equations** - Generate the differential and algebraic system of equations based on the SIMULINK blocks and variables.

[a.] Decompose Complex Blocks

[b.] Assign Variables

[c.] Generate Dependencies

[d.] Create Derivatives

Step 5: **Generate Invariants**

Step 6: **Prune unreachable locations**

While the original mission statement was to translate any MSS model to an intermediate format for use in alternative systems, some constraints currently exist with this approach. Primarily due to limitations in HSIF they have limited their approach to only support:

1. Continuous Blocks (Integrator, Zero-Pole, etc.)
2. Mathematical Operators (Excluding logical blocks)
3. Sources (Constant, In)
4. Sink (Out)
5. Switch
6. Stateflow Diagrams

The fact that the current translation method limits the HSIF model to the continuous library of SIMULINK make this method incompatible with MRICDF which has discrete time semantics.

2.3.3 Translating SIMULINK to Formal Languages

Synchronous BIP

"Behavior, Integration, Priority" (BIP) [13] is a component framework with three layers. The behavior layer consists of automata with C code descriptions of the application behavior. The Priority layer contains the scheduling rules for component interactions. Finally the Interaction layer describes the relationships between the other two layers. Synchronous BIP [24] is a subset of BIP specialized for describing synchronous systems. [1] details a methodology for translating a SIMULINK model into a Synchronous BIP model with the goal of exploiting the existing BIP validation and code generation tools. The translation is done by iteratively converting each SIMULINK block into a BIP component. Basic blocks become the equivalent elementary component in BIP, while structured blocks are traversed recursively until each of their constituent blocks have been translated. The resulting components are then composed to form a subsystem component. The translation is limited to only a subset of blocks from the discrete library due to its synchronous semantics.

Lustre

Another approach that translates SIMULINK into a formal language is presented in [66]. The SIMULINK model is translated into the synchronous dataflow language Lustre[33]. In addition to Lustre's precise semantics this approach has the advantage of granting the translated model access to the model checking [34], testing [57], and the code generation features of Lustre's commercial development suite [7]. Much like with the Synchronous BIP translation, Lustre's discrete semantics mean they only support the blocks from SIMULINK's discrete time library. Additionally, the read and write blocks, while technically part of the discrete time library, are discarded because their behavior has side effects that are detrimental to determinism in the model. Despite the discrete time limitation we believe that it is a reasonable limitation for modelling a digital controller. Our planned translation methodology is similar to that presented in [66] and will be presented in greater detail in Chapter 5. In short, typing and clock information must be extracted from the SIMULINK model and checked for errors. If they are found to be error free then that information is applied to translating blocks into MRICDF actor networks. It is our opinion that MRICDF's polychronous semantics are advantageous in a DRE environment for modeling distributed components that operate at different rates.

2.4 AADL

Architecture Description Languages (ADLs) provide yet another layer of abstraction, focusing on modeling the interactions between high level components of systems. These compo-

nents may be hardware, software, or both but in either case the contents of the components themselves are generally treated as black boxes, their implementations provided elsewhere. In this way complex embedded control systems can be undertaken as discrete tasks with different engineers separately developing component implementations that can then be integrated into the larger design modeled with the ADL. One such ADL is the Architecture Analysis and Design Language [63] (AADL). It is a textual and graphical modeling language defined by the Society of Automotive Engineers (SAE) for the purpose of creating detailed representations of embedded systems. AADL possesses all of the desired traits described above - it maps application software to an underlying execution platform and allows for user defined properties and language extensions. Furthermore it is very user friendly for the developing engineers as it has seen wide use in commercial projects ranging from aviation to space.

Components

The AADL standard is a declarative language that creates a comprehensive model of an embedded system through the use of a component based architecture. Components may be hierarchical, that is they may be composed of one or more subcomponents. Each component in the model is described by a *type* and an *implementation*. The component *type* gives the *category* of component as well as the interface *features*. Meanwhile, the component *implementation* enumerates the component's subcomponents and describes their interactions. The *implementation* also contains the properties that define the component's characteristics and requirements. These properties may be standard, user-defined, or part of a language annex extension. The available *categories* of hardware component *types* are:

1. **Processor** - The processor component *category* represents the platform for scheduling and executing software applications. Generally this means they are used to represent a microprocessor or CPU, but they may also include the functionality of the operating system - defined as a property of the processor.

2. **Memory** - The memory component *category* represents the general storage components both volatile and non-volatile (e.g. RAM, ROM).

3. **Bus** - The bus component *category* is used to represent physical communication channels and their associated protocols (e.g. PCI, VME, CAN, etc).

4. **Device** - The device component *category* is used primarily to represent physical (or simulations of physical) entities that interact with the external environment. Generally they are sensors or actuators in the system. They may also represent an off-the-shelf component that need not be modelled in detail by the developer.

The available *categories* of software component *types* are:

1. **Data** - A data component *category* in software represents a data type from the modeled application, while the data component *implementation* represents the internal structure of variables (e.g. the fields of a C/C++ struct).

2. **Thread** - The thread component *category* is the smallest schedulable unit of software, representing an execution path of code that is bound to a particular processor. A thread is always contained within a process where it is managed by a scheduler. Threads have a sequential control flow, specified either by *annex behavior* (described later) or in associated source code (callable by subprograms). Threads implementations have specified timings and dispatch protocols. Currently, the dispatch protocols natively supported by AADL are *background, sporadic, aperiodic, and periodic.*

3. **Process** - The process component *category* represents a protected address space used to isolate and protect thread memory access.

4. **Subprogram** - The subprogram component *category* represents some sequentially executable block of code (e.g. a function or method). They are called by thread components or by other subprograms.

There also exists a pair of abstract components *thread group* and *subprogram group* that serve a strictly organizational role in the model.

Finally, there is a generic component *category* referred to as a **System**. The system serves as the toplevel component in an AADL model, containing the software applications and execution platform as subcomponents. Indeed it is most commonly used as a means of grouping related subcomponents and providing a standardized interface. The *system* also has usefulness as a component for abstraction in an AADL model. It may serve as a "black box" or placeholder during early development - defining the interface and run-time properties necessary to allow the rest of the model to be tested while allowing the precise internal composition to be determined later. AADL's capability to define and use alternative implementations can allow for such *system*s to be quickly adjusted or replaced according to emerging design decisions in the rest of model.

Communication

Once the components of the model are defined, it is necessary to describe how they interact. AADL provides a few means of modelling component communication. The most commonly used method is through the use of *ports*. Ports are defined as *features* of the component type interface. Ports are directional (*in*, *out*, or *in out*) channels that transfer data or events. There are three port variants:

Event Data - Event data ports communicate event notifications and the data associated with them. These messages are transmitted asynchronously and are queued by the recipient.

Event - Event ports function similarly but only transmit the event notifications.

Data - Data ports transmit only data but do not queue messages, only the most recent message is available to the recipient.

While *ports* provide a logical description of the communication of data and events between components, *Buses* model physical communication channels in a system. An AADL *Bus* may be used to model a wide range of physical communication mediums or protocols, from PCI and CAN Buses to Ethernet or WiFi connections.

Flows are used by AADL to describe the ordering and properties of these logical paths nested throughout the system model. A *Flow* can be given as either a *Flow Specification*, a *Flow Implementation*, or an *End-To-End Flow*. The first is given in the *Flow* section of a *Component Type* definition, and describes whether a feature of that component is a *Source*, *Sink*, or merely a portion of the *Path* of that *Flow*. A *Flow Implementation* is defined in the flow subsection of a *Component Implementation*. It describes how the flow travels through the subcomponents and connections internal to that implementation. Finally, an *End-To-End Flow*, also declared in a *Component Implementation*, gives a complete path overview from the starting to terminal components of a path.

These Flows can be used to describe any type of logical flow path the designer may desire, such as data, control, or fault propagation. Properties can be specified for each flow to give information useful for static analysis, such as communication latency along a path. Such aspects make *Flows* a very valuable language component for the analysis of AADL system models.

Extensibility

AADL is extensible in a number of ways that make it flexible enough to support a wide variety of system models. One way in which AADL is extensible is through its ability to define new data types. Ultimately, though, AADL's focus is on providing an overall architecture model not a data model. Therefore, the only requirement of a data definition is that it must have an *AADL legal* name. If the name of the source data being modelled is not legal in the AADL standard then the original name can be given as a "Source_Name" property in the data's component type or implementation definition. However, more detail may be provided as required by the user. For example, using properties the size of the data may be defined in bytes. The internal composition of structured data elements such as C/C++ structs or Ada records may be defined through nesting data subcomponents. For shared data models, access and synchronization protocols can be defined as properties along with which components have read, write, or read/write access.

Much of the detail of component behavior and requirements is provided by the definition and association of *properties*. The core standard only provides four types of property values:

- Bool (aadlboolean)
- String (aadlstring)
- Real (aadlreal)
- Integer (aadlinteger)

Using base types such as these one can declare new properties, as a name and a range of acceptable values:

```
Name : type <type definition>
```

The property type defined may be a subset of one of the basic types, or define a new set of units, or combination of these. Other new property declarations may instead enumerate an ordered list of literals or make reference to a particular component or class of components. Of course, a property may also have a constant value, associating a symbolic name with a value - not unlike a constant or #define in C\C++. This ability to define new property types, values, and units of measure, is valuable when creating or applying new domain specific analyses, software tools, or enabling language extensions.

Language extensions, in particular, are AADL's preferred solution for adding modelling capabilities for features, attributes, and behaviors that are not covered by the base standard or are unable to be fully described by the definition of a new property set. In AADL, these extensions are referred to as Annex libraries. These libraries are declared inside AADL packages, with the form:

```
Annex <name> {** <content> **}
```

The name given to the annex must be a legal AADL name but the syntax used in the annex's content may make use of almost any technical language construct [29]. The only exception is '**}' is reserved to denote the end of the annex. Once defined, an annex may be employed by declaring an annex subclause in a component below the properties section (e.g. Fig 2.6 gives a generalized description of how a threads behavior may be described using the behavior annex.)

The example in Fig. 2.6 described the general syntax of the behavioral annex, but there are a number of available annexes. The previously mentioned Data Model Annex (DAnnex) [29] for example, or the Error Model Annex (EAnnex) [29], which allows for the creation of error state machines to model the occurence and propagation of error states.

```
thread Example
   features
      x, y : in event port;
end Example;

thread implementation Example.E
   annex behavior_specification {**
      <state variables> ?
      <initialization> ?
      <states> ?
      <transitions> ?
   **};
end Example.E;
```

Figure 2.6: Annex Subclause Usage

Available Tools

There are also number of tools available that support the AADL standard, its language extensions, and perform various domain-specific analyses. We will briefly discuss a couple of these that are most relevant to our work.

OSATE - The "Open Source AADL Tool Environment" (OSATE) [53] is an open source editor and development environment for the AADL standard. OSATE 2.0 is the most recent implementation of the development environment. It is built on top of the Eclipse Modelling Framework [27] (EMF). It uses UML2 [67], an EMF compatible adaptation of OMG's meta-model standard as its backend. XText [68], an open source framework for the development of domain specific languages, provides the front end functionality of OSATE's text editing environment. OSATE's textual editor supports modeling, parsing, and instantiating models in the AADL v. 2 standard. While the default distribution is strictly a text editor, the environment is extensible through the use of Eclipse plug-ins.

These extensions include official plug-ins such as ARINC 653 [12] and the previously mentioned Error Model Annex. There are also a number of third party plug-ins such as the Behavior Annex and a graphical model editor [8].

Cheddar - The default OSATE framework excels at detailed modeling of system composition and provides basic analysis capabilities to check for model coherence, resource usage and statistics. For more detailed analysis, particularly of nonfunctional and run-time properties, there are some commonly used plug-in tools, such as Cheddar [65]. It is an Ada based framework for real-time schedulability analysis of AADL models. Cheddar has built in support for a variety of known scheduling algorithms, as welll as a defined Ada-like extension language to add new temporal behaviors that are not already covered.

2.5 Related Works - Model Analysis and Code Generation Techniques for AADL

2.5.1 Ocarina

AADL's capability to model the hardware and software components of a DRE concurrently and its extensible property set for the description of both functional and nonfunctional attributes or requirements allow for precise specifications. Its component reuse, refinement, and inheritance capabilities lend themselves to the desired iterative design approach. What AADL lacks is an automated step from the final prototype to a fully realized system. Ocarina [36] addresses this by providing a tool for rapid prototyping through the generation of high-integrity code from AADL models. This code is generated to run atop a custom high-integrity middleware called PolyORB-HI[71]. Ocarina is built using a subset of Ada described by the Ravenscar Profile as suggested in [2]. This subset was chosen for its guarantee of schedulability and safety properties.

The user manually builds their DRE using their choice of AADL standard. The Ocarina frontend is modular and can be configured to support either the AADL v1 or AADL v2 standard. The model is fed into the appropriate lexer to tokenize the textual model. The tokens are parsed to ensure adherence to AADL's grammar standards. If any errors or warnings are detected then they are displayed to the user at this point, otherwise an Abstract Syntax Tree (AST) representing the model is generated. Error-free ASTs are handed off to a semantic analyzer to be scanned. During its resolution phase, the semantic analyzer performs some value substitutions to simplify the model before determining whether the semantics conform with those of AADL. After this, the model is instantiated according to the rules found in [63] to create an *AADL Instance Tree*. This tree is then checked for any incoherence, such as any missing required subcomponents, necessary property definitions, or disallowed protocols. A coherent instance tree is the final product of the ocarina frontend (Fig. 2.7).

The Ocarina backend (represented in Fig. 2.8) is also modular. Each module represents a possible target language for the generated code. The first step of the backend is an expansion of the *instance tree*, simplifying its structure and annotating it with additional information necessary for the code mapping process. This *expanded instance tree* is then transformed using the syntax in the module of the targeted language to build an *intermediate syntax tree*. The *intermediate syntax tree* is scanned in conjunction with the PolyORB-HI middleware to generate the final code with hooks into the appropriate middleware services. This generated code, as well as any attached source code from the user, is tested to ensure it meets the restrictions of high-integrity systems as described in Annexes D and H of the Ada 2005 standard [38].

Once the code has been generated it must be deployed in a way in which it can be compiled and executed for the target distributed platform. Ocarina chooses to deploy statically [36]

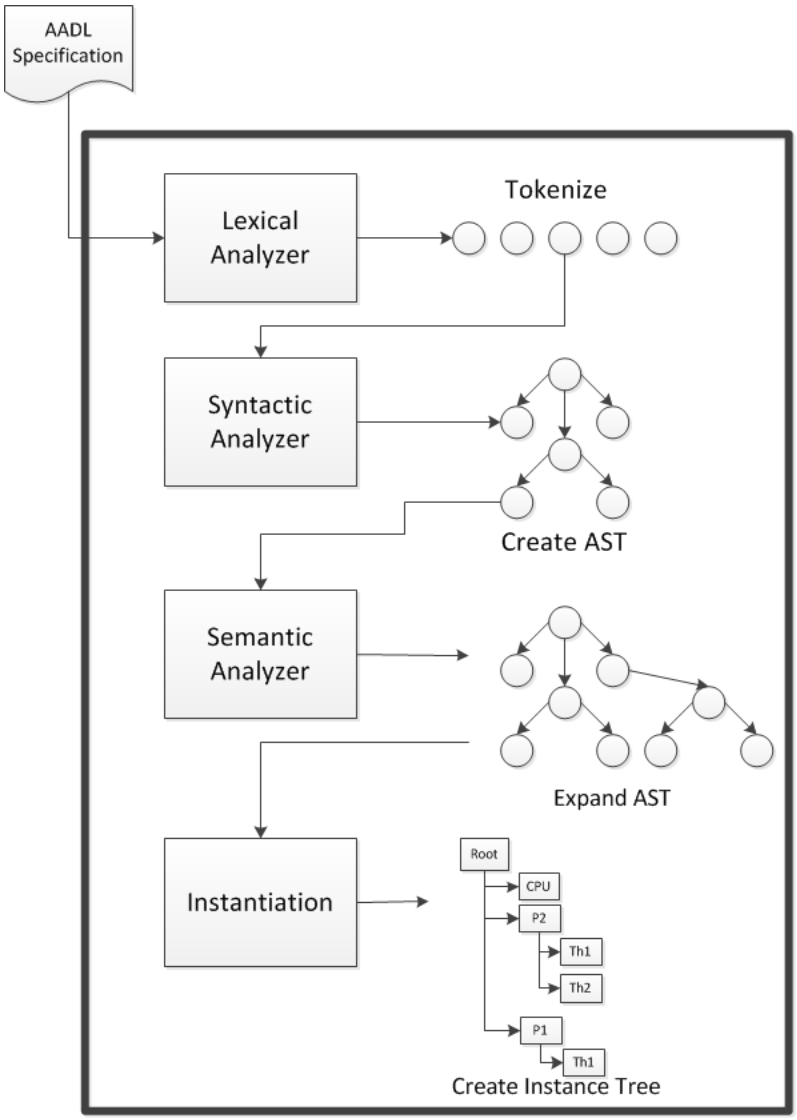

Figure 2.7: Ocarina Frontend

rather than dynamically, analyzing the model at compile time to determine the necessary PolyORB-HI components and their desired properties. While this design choice requires the addition of an analysis phase, that process can be automated and it avoids the potential safety pitfalls present in a dynamic deployment - such as dynamic binding and memory allocation.

The structure of AADL models is well suited to automating the static analysis process, as it can already be thought of as detailing the deployment view of the software. Nodes of the deployed software are analogous to *process* components in AADL. The tasks of each node are given by the *thread* components, and necessary detail and model refinement derives from those components' properties. The locations of these nodes in the distributed system can be

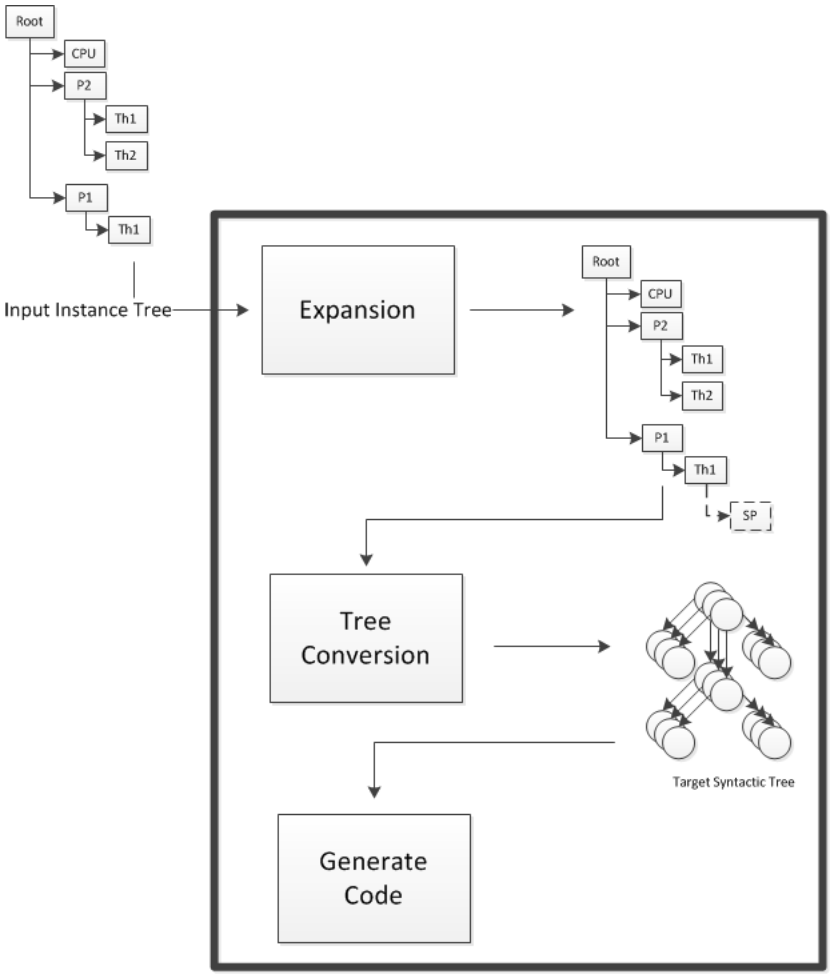

Figure 2.8: Ocarina Backend

determined by their bindings to the *processor* components of the hardware portions of the model with *buses* and *ports* to describe the interconnections. Communication between the nodes of the deployed code is handled using a system of sender/receiver tuples in the glue code, a detailed description of this approach can be found in [36].

2.5.2 Additional AADL Based Methodologies

AADL's flexible, high level approach to MDE provides an efficient basis for the modeling and analysis of complex embedded systems. This fact, coupled with AADL's widespread use in the automotive and aerospace domains, has led to research into a number of frameworks that would aid in formal translation and validation of AADL models or automating code generation.

Translation

A variety of model translations have been proposed. For instance, the Fiacre [17] project is a TOPCASED [56] based approach that aims to provide a common linguistic pivot point for ADL analysis. Fiacre serves as an intermediate language into which modeling languages such as UML, SysML, and AADL may be converted so that they may be analyzed in model checkers CADP [32] and TINA [16]. For AADL, thread behavior is specified using the behavioral annex. The work in [13] proposes the translation of AADL into BIP [1]. The translated model can then be simulated, analysed, and generated into C/C++ code using BIP's tool suite. Meanwhile, TASM[69] seeks to transform AADL into Timed Abstract State Machines in a way that is provably semantics preserving [18].

Language Extensions

Unlike the preceding works that proposed managing AADLs lack of formal semantics through translation, Compass[25] proposes a new formally defined language derived from AADL. The System-Level Integrated Modeling language (SLIM) is an AADL derivative that describes hardware and software behavior. Unlike its predecessor it has been extended with model behavior and an enriched version of the Error Model Annex [54]. These enhancements allow it to provide a threefold specification of the model: its nominal behavior, its error behavior, and fault injections using probabilistic models to inject faults coupled with SAT-based and symbolic model checking. Compass analyzes the system behavior under nominal and simulated failure conditions to determine the reachability of errors states. The work done in [26] continues the development of Compass by applying slicing techniques [52] to reduce the state space explosion encountered when model checking SLIM specifications.

Polychrony

INRIA has been developing their own polychronous tool chain for AADL [47, 48, 70]. In this ongoing work the entire AADL model is translated into Signal by translating from ASME (AADL Syntax Model under Eclipse) to SSME (Signal Syntax Model under Eclipse) as a java eclipse plug-in. The translated model accounts for the temporal (thread schedulability, communication delay) and behavioral aspects of the original system. Originally, the behavior needed to be specified in the behavior annex but [70] added support for generating Signal code from a subset of Simulink using Gene-Auto[59]. Additionally, their work in [19] proposes an extended version of the Behavior Annex. They propose adding synchronous aspects of event triggered guards for state transitions. Such guarded transitions are associated with actions that occur during the transition. The goal of this is to assist in bridging the gap between AADL and Signal behavior for translation. There also exist tools built entirely toward generating executable code from aadl specifications. One is, of course, Ocarina as it forms the basis of our toolchains code generation, albeit extended to support a polychronous

specification. Another is the platform dependent software generation proposed in [29]. The AADL model describes the target system, annotated with properties that describe the capabilities and requirements of the target. Using these properties the tool generates code specific to the target platform by drawing upon a library of C/C++ code snippets. These works provide the convenience and expedience of automated code generation but lack the formal verification capabilities of the other approaches. Our work does not deal with fault or probabilistic models as done with the Compass project. However, it has the advantage of being compatible with standard AADL models. Furthermore, we can leverage MRICDFs polychronous model for analysis and multithreaded code generation. Similarly, we believe this gives an advantage over the other formal translations such as those in BIP and Fiacre. Indeed, a polychronous approach is more naturally suited toward modeling a distributed system wherein the components are operating at different rates. Unlike the other polychronous approaches, we do not seek to translate the entire AADL model for verification or simulation. Instead we focus on creating an approach that can be integrated as part of an end-to-end toolchain.

Chapter 3

Formal Language Preliminaries

3.1 Formal Languages

Low level programming is capable of implementing a large class of applications for real-time embedded systems. As a result its unsurprising that it is commonly known and widely used for this purpose. While many real-time services and platform features aren't natively accessible in these languages, a number of Application Programming Interfaces (API) have been created to provide them with a standardized means of interfacing with these features. A prime example of this is the POSIX threads API for C/C++ to provide specialized features for multithreading controls. However, while these APIs allow developers to support such a diverse class of applications with low level languages, they still suffer from a number of drawbacks. For instance, modern embedded systems frequently employ a significant amount of concurrency both between interacting threads on a processor, other components within the distributed system, and with the larger environment. Even with the aid of a specialized API such as POSIX the manual implementation of these concurrent features is both tedious and error prone. The developer must, in effect, manually schedule the time triggered communication aspects of the system and any communication between parallel threads risks deadlock or creating race conditions. Testing and verifying that the system is free of such faults then becomes an additional cost. Second, a low level approach makes it difficult to observe a relationship between a section of code and a component or constraint in the initial specification. This ambiguity makes it difficult to ascertain whether an element of a thread can be safely changed or whether it is fulfilling a requirement of the design. Indeed these threats to the safe and secure execution of the system posed by concerns of thread safety, as well as the potential for error present in manual implementations, illustrate part of the need for verification. Yet the correctness of these systems is difficult to determine manually or even through automated verification tools due in large part to the aforementioned ambiguity in these low level languages. Even with automation the state space required to exhaustively test a system will quickly become prohibitively large and can lead to overlooked errors.

3.1.1 Synchronous Languages

It should come as no surprise then that some designers have looked to higher levels of abstraction as a solution for simplifying the problem of identifying critical sections and validating the behavior of the system. These languages are still primarily used for early development. Their abstraction is well suited to quickly creating prototypes for high-level specifications that can be verified and simulated. Some, however, have been designed with the goal of supporting the entire process from initial specification through to implementation. These tend to rely on automatic code generation to provide the low-level code from the abstract specification. The correctness of the generated code is guaranteed. This coupled with the simplified nature of these languages reduces the duration required for development. The synchronous approach [14] is one such route taken to implement a high-level approach. It is based on a mathematical foundation that simplifies the system by abstracting away real-time concerns and replaces them with a strictly logical notion of discrete instants. The core notion of this abstraction is referred to as *synchrony*, or the synchrony hypothesis. In general terms, the synchrony hypothesis divides time into discrete instants between which no events or computations occur. Working on this assumption the programmer does not need to be concerned with timing of the computations occurring within an instant, merely assuming that they will complete before the next instant begins. This offers a platform-independent model of system behavior that is valid as long as the final system is fast enough to satisfy the assumptions made by the synchrony hypothesis. We examine four of the synchronous languages. Two are strictly synchronous, Lustre which uses a data-flow declarative approach and Esterel which has a control-flow, imperative approach. The other two are polychronous, both data-flow declarative approaches: Signal and MRICDF. While similar in approach, the two differ in terms of implementation and approach to model analysis. In the next section we will provide some preliminary background on the synchronous approach and contrast its different languages to illustrate its uses as well as our reasoning for supporting its adoption in our methodology.

Preliminary Definitions

Here we informally introduce some concepts that are necessary to understand the fundamentals of the synchronous languages. As previously mentioned, these languages are all built around a fundamental assumption called the *synchrony hypothesis*. This core concept means that physical time is abstracted away from the model, allowing the developer to focus on the logical behavior of the system. These systems are also reactive, the arrival of new values on an input represent an *event*. When an event occurs it triggers a computation that must be completed before the next event occurs. Instead of tracking the duration of the computation in physical time, the behavior is divided up into discrete, logical *instants*. These instants represent the boundaries of execution for each such reaction. There is no relation between these instants and the physical time taken for the computation. Rather the only concern is

respecting the assumption made by the *synchrony hypothesis* and ensuring that the reaction to any given event is completed before the end of the instant it triggered. A *signal* is the *totally ordered* sequence of events that flow from a source to a sink interface port in the system. Events that occur within the same instant, regardless of their precise physical time, are thus said to be *synchronous* with each other.

Several synchronous programming languages have been implemented around these core principles. Each has its own quirks in terms of specification, compilation, and intended applications but they all share some common attributes. In addition to being event-driven reactive systems, each language is *synchronous*. By this we mean that each reaction initiates a batch of operations that are executed within a common instant. Some implementations are *monochronous*, enforcing a single master clock, while others are polychronous and only enforce synchronicity between components when they must communicate with each other. This approach models *concurrency* within the system specification at a high level. As a result, statements that are determined to be independent of each other will result in generated code that executes in parallel. Perhaps most importantly, the languages all guarantee *deterministic execution* because all computation and communication will be completed within a single instant. Thus for any given triggering event, the subsequent output for the end of that instant can be predicted. With these properties in mind we will now discuss four of the synchronous formal languages beginning with the monochronous approaches and then the polychronous models.

Esterel

Esterel[15,23] with its beginnings in 1984 is the oldest of the synchronous languages and well known, having been used in a number of industrial projects in Europe. As such its perhaps not surprising that it bears the greatest resemblance to the general purpose programming languages that preceded it. Esterel's style and syntax is that of a textual, imperative language. Its focus on specifying the control of the target system also makes it fairly unique among the synchronous languages and also lends a feeling of familiarity to those with a background in standard software development.

Signals in Esterel serve as both inputs and outputs. A signal may be present or absent during a given instant as previously defined. Further, the signals may or may not have a value associated with them. Signals without a value are referred to as *pure signals*. *Tick* is an example of an important pure signal in the language, its an implicit part of all Esterel programs. Tick is present every instant and represents the global abstract clock that drives the system. Each occurrence causes all threads to resume their execution from where they last paused. Signals communicate by *broadcasting* their presence (and potential value) through the use of an *emit* statement. Emitted signals are visible instantaneously throughout the system. For coherence purposes, signals may either be present or absent but never both.

The basic entity of Esterel is a *module*. Modules have an *interface* that defines their input

Table 3.1: Esterel Statements

Statement	Description
p;q	Statement Sequencing - execute p followed by q
p || q	Execute p and q in parallel until both have terminated
emit S	Make signal S present
emit S(e)	Make signal S present with the value e
present S then p else q end	If the signal S is present then execute p otherwise execute q
x := y	Variable assignment
if e then p else q	Data Conditional Statement
loop p end	loop indefinitely unless preempted
abort p when S	Run p until but not including the instant signal S becomes present
pause	Do nothing in this thread until the next instant
halt	Do nothing until preempted

and output signals and a *body* that is made up of statements like those shown in Table 3.1. An example module is shown in Fig. 3.1.

```
module ABRO:
input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O
each R

end module
```

Figure 3.1: Esterel ABRO

ABRO is a simple module that waits until both A and B have arrived and then it emits the pure signal O, it resets each time R arrives. It concisely exemplifies Esterel's emphasis on control flow, pairing iterative statements like *loop* with synchronous concepts like those involved in parallel reactive behavior of *await*. ABRO also displays Esterel's capacity for preemption and delay. Preemption is performed through the use of special statements, in this case the *loop* preempts and reinitializes the program every time R occurs. These statements can be composed either sequentially, using '*;*', or in parallel, using ||. With sequential statements$(S_1; S_2)$ the first statement must terminate before the second can begin. Parallel composition meanwhile, $(S_1||S_2)$, executes S_1 and S_2 simultaneously and does not terminate until both statements are finished. In ABRO this can be seen as the module waits for

both A and B, in parallel, to arrive before sequentially proceeding to the emission of O. In addition to the basic statements mentioned previously, Esterel has some derived statements such as *await S* which is actually syntactic sugar applied to the commonly used statement *abort halt*; *when S* as a shortcut. Esterel is not, however, considered a complete language in its own right. It supplements its otherwise capable framework by calling on external functions to fill in the gaps within its functionality and data types. The internal behavior of these external functions is treated as a black box by the compiler. This is a fairly common feature of synchronous programming that provides the option to quickly extend the capabilities of the language by calling external sources in standard, informal code but at the cost of reducing the verifiability of that code segment to its interface and mere assumptions made about its performance.

Compilation in Esterel concentrates on proving that each reaction is a deterministic function with a unique solution. That is, the specification should not deadlock due to cyclic dependencies nor should it have multiple potential solutions. The existence and validity of a solution in Esterel is reasoned about through the exploration of the set of control states of the program, with the instants being delineated by *halt* statements. Being a *monochronous* language, the clocks of Esterel signals are all tightly related to an assumed global reference clock. This clock, represented in Esterel as the pure signal *tick*, is present in each instant of the system's execution. Therefore, the rate of *tick* is relative to each of the signals in the specification, such that it must have the highest rate and each other signal's clock can be described as a strict subset of *tick*'s. In a distributed design, therefore, each component's reaction instants are a subset of the master clock's. Because of this tight relationship, changes to the signals of a component may change its speed relative to the global clock, which in turn may require resynchronization between the other components in the system and *tick*. Another side effect of this approach is that code generated from a monochronous specification for such a distributed design can suffer from inefficiencies that result from being driven by a single master clock. Although each occurrence of *tick* signifies the start of a new instant, when dealing with multiple subcomponents it is quite possible to encounter a scenario in which none of the signals of a component will have a reaction within a given instant. This means that the code generation from such a model will unnecessarily wake some components with *tick* each instant to check for other present inputs although they may have none.

Lustre

Unlike Esterel, Lustre [33] is a data flow oriented, textual language. The fundamental blocks of Lustre are referred to as *flows* and *nodes*. In Lustre all variables are flows which are potentially infinite sequences of values. In addition to its value sequence, each flow is also associated with a clock. Constant variables are flows that have the same rate as the *basic clock*. The basic clock serves as the global system clock, much like *tick* in Esterel. It is present in every instant during the execution of the system, therefore it has the highest rate of all the system clocks. There may be slower derived clocks in the system through the use

of conditional undersampling, but all can be related as a subset of the base clock.

A variable's type is given at its declaration. The types available in Lustre are limited to either integer, real, or boolean. If more advanced types are needed they can be imported by calling on an external source language, similar to how Esterel calls on external functions to supplement its functionality. Variable values are determined by an assigned expression which is composed of constants, operators, and other variables. Each variable is defined in this fashion exactly once. The operators are defined pointwise so that they may operate on infinite flows, so that for a given n^{th} instant of execution the operator returns the n^{th} value of the flow. Lustre supports standard operators for arithmetic, comparison, and conditional statements. In addition to these basic operators, Lustre has four special temporal operators shown in table 3.2 along with example flow values.

Table 3.2: Lustre Statements

tick	1	2	3	4	5	6	7	8	9
x	0	1	2	3	4	5	6	7	8
pre(x)	nil	0	1	2	3	4	5	6	7
$1 \rightarrow$ pre(x)	1	0	1	2	3	4	5	6	7
B	T	F	F	T	F	F	T	F	F
y = x when b	0	nil	nil	3	nil	nil	6	nil	nil
z = current(y)	0	0	0	3	3	3	6	6	6

Pre(x) allows recursive definition of variables by generating a flow that is equivalent to the flow x, except delayed by an instant. The first value of this new flow cannot be inferred and so defaults to *nil*, unless the $\rightarrow$ operator is used to insert an initial value. The *when* statement is used for conditional undersampling. X when B generates an output flow that is a subset of the flow of x, only when B is 'true'. Finally, *current* allows for a flow with a slower clock to be sampled by a faster flow through oversampling. The current(x) operation returns the value of x's flow the last time is was present. Nodes are functions of flows, they define an interface of input and output flows and a body that is a set of equations. In this way a Lustre model is a system of equations. We present here in figure 3.2 the previously described ABRO program as it is implemented in Lustre.

The effect of Lustre's dataflow focus, as opposed to the control flow oriented Esterel, is immediately apparent in this new implementation. Lustre lacks control statements such as synchronous iterative loops or globally broadcast signaling to provide concise structure to the program. Instead the output flow, O, is defined as a function of the current and past values of the input flows. EdgeA and EdgeB variables track their respective inputs. They initialize to false and hold that value until their associated input arrives with a true value. They hold their previous value through the use of the *pre* operator. An occurrence of 'true' on R will force *edgeA* and *edgeB* back to false. O, meanwhile, calls the node *edge*, which

```
node edge (x : bool) returns (y : bool);
let
    y = false → x and not pre(x);
tel

node ABRO (A, B, R : bool) returns (O : bool);
    var edgeA, edgeB : bool;
let
    O = edge (edgeA and edgeB);
    edgeA = false → not R and (A or pre(edgeA));
    edgeB = false → not R and (B or pre(edgeB));
tel
```

Figure 3.2: Lustre ABRO

returns 'true' only on a $false \rightarrow true$ transition of the pair of inputs.

The Lustre compiler performs *clock checking* to determine whether the clock constraints of the system are satisfied. Specifically, most operators require that all their operands have the same clock rates as they can only operate when all the operands are present. Variables are checked to ensure none are defined more than once to protect system coherence. Finally, Lustre requires that all specifications are entirely devoid of cycles, which are checked for during *causality analysis*. This guarantees well behaved determinism but it can cause some specifications to be rejected erroneously. If the model passes the causality analysis without any detected errors then the compiler proceeds to code generation. The nodes of the specification are first expanded recursively to achieve a flat program hierarchy. This is necessary to provide context for the flows so that they can be scheduled deterministically for the generated code. Then a single infinite loop representing a single execution cycle of the program is constructed. The order of the computations is determined by dependencies found in the flattened model during causality analysis.

3.1.2 Polychrony

Signal

Unlike Lustre's *basic clock* or the pure signal *tick* in Esterel, SIGNAL [46] does not assume the existence of a global reference clock. Signal is a textual, data-flow oriented language but unlike Lustre, which defines each variable(flow) as a function of the values of its input flows, SIGNAL takes a relational approach where each statement or component creates constraints. Thus, *Relations* are the core of the SIGNAL language. The system's behavior is modeled as sets of relations between signal flows. As previously mentioned a key difference between this

approach and those of the functions described in Lustre is that SIGNAL's relations imply both *functional constraints* on the values and *temporal constraints* on the clocks of the involved signals. Further, the way in which an output signal is used will implicitly constrain the behavior of its component input signals. Relations in SIGNAL have the general syntax of $< id >:=< expr >$. The expressions can then be partitioned into one of two operational sets - *monoclocked* or *multiclocked*. All signals that are involved in a monoclocked operation are implicitly synchronous with one another. Signals involved in a multiclock operation, however, may have different clock rates. Now we will describe the operational primitives used in SIGNAL expressions and their implicit clock constraints.

Functions (Monoclock) Functions define instantaneous relations between signals $S_1...S_2$. All input and output signals of the function are, as previously mentioned, implicitly required to be synchronous. A function in SIGNAL is an output signal, or variable, that is assigned the result of an operation. The syntax of these functions is given in Fig. 3.3 while a listing of available operators, by type, is likewise shown in Fig. 3.3.

$$S_O := F(S_{i1}...S_{in})$$

Figure 3.3: Signal Function

Table 3.3: Signal Function Operators

Type	Operators					
arithmetic	+	-	x	xx	/	modulo
comparison	=	/ =	>	>=	<	<=
boolean	not	and	or	xor		

Delay - $ (Monoclock) The delay operator ($) acts as a buffer, shifting values to later temporal indices. Initial values must be provided for each buffered position. In this way, delay allows the user to access past values of a signal and aids in protecting against uninitialized data access.

$$S_O := S_i \text{ \$ n } init \text{ } [c_1...c_n]$$

Figure 3.4: Signal Delay

In Fig. 3.4 we see the general form of the delay statement. N is the size of the buffer while c_1 through c_n are constants of the same type as S_i and S_o.

Undersampling - 'when' (Multiclock) The keyword *when* functions much like its Lustre counterpart, The result of this relation is that s_o is assigned the value of s_i only when the condition is both present and true, otherwise s_o is absent.

$$S_O := S_i \text{ when } < cond >$$

Figure 3.5: Signal Undersampling

Because this is a multiclock relation, the abstract clocks of the involved signals may not be identical. The clocks of an undersampling relation are given in Fig. 3.6 where $[S]$ denotes a signal is present and true.

$$\hat{}S_O := \hat{}S_i * [cond]$$

Figure 3.6: Signal Undersampling

Essentially, the clock of the output signal is the intersection of the clock of the input signal and instants for which *cond* is true.

Priority Merging - 'default' (Multiclock) The keyword *default* allows the user to merge the flows of two signals. This is done deterministically, so that the value of S_1 takes priority so long as it is not absent. Otherwise, the value of S_2 is propagated.

$$S_O := S_1 \text{ default } S_2$$

Figure 3.7: Signal Merge

The clock of S_O is therefore the union of the clocks of the signals S_1 and S_2. SIGNAL groups

$$\hat{}S_O := \hat{}S_1 + \hat{}S_2$$

Figure 3.8: Signal Oversampling

sets of relations as *processes*, much like Lustre's *nodes*. As seen in the ABRO example of Fig. 3.9, a process defines an interface of input and output signals as well as any parameters needed for initializing constants and a body composed of relational expressions. There are also two operations specific to processes. The first is the composition operator "|". This operation is equivalent to expressing a conjunction of the behaviors of the involved processes $P_1|P_2$. The operation is both commutative and associative and the composed processes communicate via common signals. Input signals in P_1 may be outputs in P_2, or vice versa, but due to data flow's *single definition principle* each signal may only be defined, e.g. appear on the LHS of an expression, once. The other capability unique to processes is the definition of local signals. An example of this is in the SIGNAL ABRO specification. Location signals are declared in a $where < type >< ID > ...end$ block. Locally declared variables are limited in scope to the process in which they were declared.

The ABRO example presented here bears some resemblance to that shown for Lustre. Both take a data flow driven approach to specifying the system. Note that unlike Lustre this version starts by explicitly asserting certain clock constraint relations between the different signals of the specification. This is necessary later during compilation when SIGNAL

```
process ABRO = (? event A, B, R; !event O;)
(| A_Received ^= B_Received ^= after_R_until_O ^= A ^+ B ^+ R
 | A_Received := not R default A default A_Received $ init false
 | B_Received := not R default B default B_Received $ init false
 | curR := not O default R default preR
 | preR := curR $ init true
 | O := when A_Received when B_Received when preR
 |)
 where
    boolean A_Received, B_Received, curR, preR;
 end
```

Figure 3.9: Signal ABRO

determines whether the specification is *endochronous*, having a single master clock that is present in each instant, and therefore capable of sequential code generation. A_Received, B_Received, and preR are all locally declared signals that maintain the state of the system by tracking whether their associated input signal has arrived. The output, O, is an event (similar to a pure signal in Esterel) signal whose clock is the intersection of the clocks of the true values of these three state variables.

Lustre's approach to parallel composition was to expressly forbid any delay-free cycles, regardless of reachability. Esterel, meanwhile, allows so called "false causal loops" and cyclic dependencies that are found to reside in unreachable states although this comes at the cost of having to prove that these specifications have a unique fixed-point solution. The polychronous approach taken by SIGNAL instead accepts a set of signal relations and the constraints they imply. From these relations and constraints it may find zero, one, or multiple solutions. This additional flexibility, where all three scenarios have accepted semantics, can be useful for prototyping and high-level specification - although sequential code generation still requires the existence of a unique solution as in Esterel. the first step in SIGNAL compilation is recursive expansion of relations until the process is entirely composed of primitive statements. From the newly expanded process, clock constraints (implied and explicit) are extracted in order to synthesize a clock hierarchy and determine whether or not a process is *endochronous*. An endochronous process has a single *master clock*, which is a clock that when it is absent then no other clocks are present in that instant of the process. For processes determined to be endochronous, sequential code can be automatically generated using extracted clock constraints and data dependencies obtained from the specified relations to determine a static scheduling of tasks. SIGNAL also has access to a model checking tool Sigali[45], that converts the process specification into a Polynomial Dynamical Equation System (PDS). The PDS represents the state of boolean signals in "integer modulo 3" space: $presentandtrue \rightarrow 1$, $presentandfalse \rightarrow -1$, and $absent \rightarrow 0$. Non-boolean signals are simply either present (1) or absent (0). Thus encoded, Sigali can mathematically rea-

son about the reachability, invariance, and attractivity of the specification without having to fully enumerate the system's state space as would be needed in an automation based approach.

MRICDF

MRICDF is a data-flow formalism that provides a graphical interface for synchronous language development. The polychronous model of computation used by MRICDF is similar to that used in Signal, albeit with a different approach to epoch analysis. There are four primitive actors in MRICDF from which any other necessary actor network can be built. They are:

• *Buffer:* This actor emits the value input in the previous instance, essentially delaying its output signal by an instant. It takes a single input and emits a single output. The buffer must be defined with an initial value to emit when the first input is received.

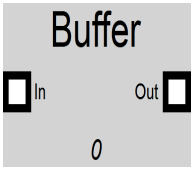

Figure 3.10: $BufferActor$

• *Sampler:* This actor takes two inputs, the first is of a type of the developer's choice, while the second must be a boolean. The sampler emits a value on its single output port only when there is an event on the first input and the second input is both *present* and *true*. The output rate is the intersection of the input rates.

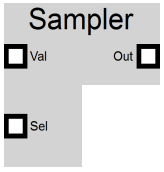

Figure 3.11: $SamplerActor$

• *Prioritized Merge:* The merge actor has two inputs, emitting a value on its single output whenever an event occurs on either of them. however, should both inputs have an event occur during the same instant, the value of the first input will always be chosen to be emitted on the output port. The rate of the resulting output signal is therefore the union of the rates of the input signals.

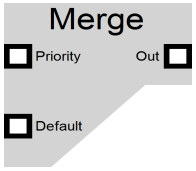

Figure 3.12: *MergeActor*

• *Function:* Finally, the function actor which supports a user selectable number of input and output ports. The function's n inputs are implicitly synchronous with one another. A user defined function, written in C, is applied to the inputs to synchronously generate all m outputs in the same instant. The internal operation may be any arithmetic or boolean computation.

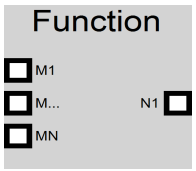

Figure 3.13: *FunctionActor*

Each also has known epoch and boolean equations associated with them that are used in the analysis and code generation steps, they can be found in [50]. In addition to these primitives, composite actors may be defined and saved for later use. The composite actors are composed of any number of primities or other composites, encapsulating a specific task behavior for reuse.

The process by which the actor network model is analyzed and prepared for code synthesis is referred to as *Epoch Analysis*. This analysis is carried out by *EmCodeSyn*[40], the graphical development environment for *MRICDF*. First, the actor network is transformed into a system of boolean equations based on the equations associated with each actor as previously described. The next step is to make sure there are no dependency cycles within an instant, the presence of which leads to *deadlock* situations. This happens if there are any looped signals in the network that do no have at least one *buffer* actor in the loop. Assuming no deadlock situations are identified, its necessary to determine whether the code to be generated is sequential or multithreaded. The generated code will be sequential if a *Master Trigger* can be identified. The *Master Trigger* must be a signal that is present in each logical instant. If more than one such signals are found to exist then they must be synchronous with one another, precedence and data dependence is used to decide which of the candidates has primacy over the others. The details of how a *Master Trigger* are determined are discussed in [41], essentially the system of boolean equations is checked for a *unitary positive prime implicate* using SMT-based techniques. If no single such signal can be identified then a sequential implementation is infeasible without using *exogenous constraints* to synthesize a *master trigger*. Making such a modification would change the original model, so instead the next step is to look for *Partial Triggers*[51] that would enable a multi-threaded implementation. A *Partial Trigger* may be absent in some instants, but there must not be

an instant in which all *Partial Triggers* are absent. Each *Partial Trigger* is effectively a *Master Trigger* for an individual thread.

After the model has been screened for deadlocks and its trigger or triggers have been identified, *EmCodeSyn* can generate C code from the actor network. Three files are generated: a *main* file that contains the scheduling order determined by the epoch analysis, a *header* file with the definitions of all the generated functions and primitives, and a *function definition file* that contains as the user defined code for *function* actors. For sequentially implementable code, computations are handled as the primitive actors interacting during each round of a loop. The primitive actors are replaced by equivalent blocks of C code. In the multithreaded situation, a thread is first generated for each discovered *Partial Trigger*. The clock trees of the *Partial Triggers* are traversed depth first, populating each thread as with the single sequential code. During the traversals, nodes of the tree are checked to determine whether they had been tagged as being present under multiple root nodes (*Partial Triggers*), because this indicates the node represents a variable shared between threads. Any nodes identified as shared variables have *wait-notify* constraint code automatically generated around them for thread safety. This capacity for correct by construction generation of thread parallelism and synchronization objects plays a crucial role in the code generation model used by APECS as described in Chapter. 4

Chapter 4

APECS Methodology

4.1 Motivation

As discussed in Chapter 1, dealing effectively with distributed real-time embedded (DRE) system development requires the creation of a dedicated methodology that can handle all the levels of design concerns. The methodology must be able to encapsulate all of the system's resources (processors, memory, etc) and model their allocation. Further, the methodology should make explicit the behavioral expectations of the system (control/data flow, modes, safety requirements, etc). It is critical that these concerns are addressed so that designers may be able to detect inadequacies within the hardware specifications or architecture hierarchy early in the design process to minimize the costs of changes. Model Driven Engineering (MDE) is a convenient and commonly taken approach to address these concerns. In Chapter 2 we presented the state of the art in terms of general industry standards for systems modeling languages. While each language is possessed of its own unique strengths and weaknesses there were a few key commonalities. None of the modeling languages are programming languages nor are they directly associated with a particular methodology for progressing from high-level specifications in their standard to implementable software and hardware models. As a result it is necessary for the stockholders and designers to create or appropriate their own tools and process suitable to their domain. Additionally, while all are capable of specifying the structure of a system and the the flows through that hierarchy, they lack the semantics and the capacity to describe the implementation of the behaviors of those components in a formally verifiable way. Recall, for instance, the requirements specification given in Figure 2.4. While a helpful reference marker for a developer, it provides the requirements in natural language absent means of enforcement. In chapter 3 we introduced the formal synchronous languages which can help alleviate these issues with formally verifiable code generation. This chapter presents our methodology and tools designed to address the concepts and implementation details needed to enable distributed, multi-threaded code generation from formal software specifications for modeled critical systems.

4.2 Approach

APECS is a top-down, model based approach that combines an AADL foundation that supports the specification, design, and verification of systems with concepts of polychronous languages to provide formal behavioral semantics and the automated generation of parallel, executable code. The goals of APECS are therefore to:

- Capture the structure and composition of complex systems
- Integrate polychronous software methods with the modeling and code generation techniques
- Support modularity, reuse, and iterative design refinement.

In order to achieve these goals we propose a top-down methodology (Fig. 4.2) based on iterative refinement and formal verification.

- Initial Requirements Specification
- Defining the Required System
- Definition Refinement
- Model Optimization
- Validation and Code Generation

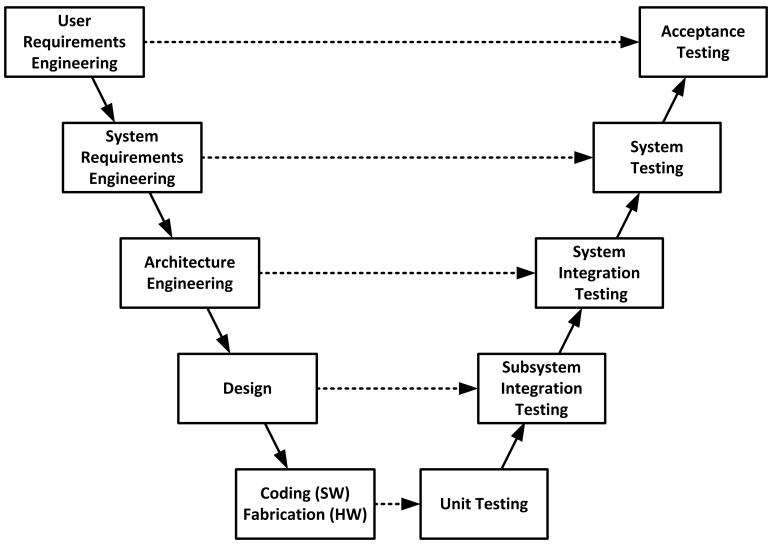

Figure 4.1: Vee Life Cycle [30]

This approach is similar to the traditional "Vee" development lifecycle (Fig. 4.1) iteratively applied at each step as it is refined. APECS uses a model based approach to capture the structure and resources of the model during this process, while the behavior is specified separately and associated with the corresponding model component. Now we will provide a summation of each phase of this approach.

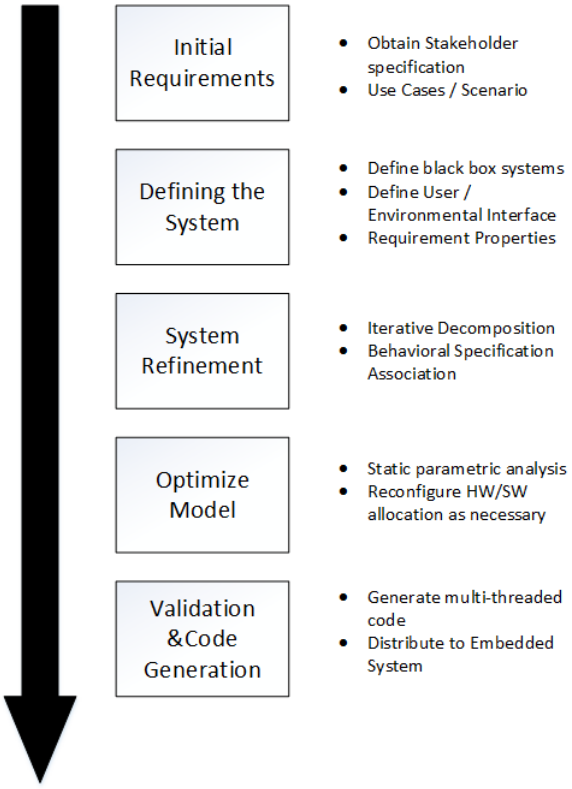

Figure 4.2: Methodology Phases

Initial Requirements Specification

At this step the stakeholders in the system provide their requirements for the deliverable system. They provide the objectives, top-level use cases, and areas targeted for critical activities. In our methodology we assume this is something provided at the beginning of the project. While we intend to eventually generate a formal system model, at some point in its earliest form the project begins as an informal statement of objectives and features.

Defining the Required System

This phase is for the specification of the system at a high-abstraction level. The intention is to define just the top-level, mission critical components. These components are treated as

black boxes. Their interfaces are defined per the initial requirements specification and their required properties are added as component annotations. Using an ADL based approach, this step consists of creating generic system and device types that can be instantiated with more specific details as the model matures.

Definition Refinement

The top-level systems are decomposed and populated with subcomponents that fit the requirements of the parent system. This process is carried out iteratively as each system is decomposed and in turn its subsystems are decomposed until the model is fully flesh out down to its lowest tier of components. During this phase some systems that were serving as black box placeholders may be replaced by more specialized model components. For instance a generic system may be replaced by a device representing a specific off-the-shelf component. Additionally, behavior specifications are associated with their corresponding components are they're defined.

Model Optimization

At this phase the system hierarchy has been fully defined. With the full scope of the system's resources and properties available, static analysis techniques are employed and different configurations, hardware bindings, and components instantiations are tested to improve upon and balance performance with the production costs of the final product. This optimization step and the previous refinement step are repeated until the designers are arrive at a satisfactory solution.

Validation and Code Generation

A number of verification processes are applied using the tools available to perform static analysis such as real-time and schedulability analysis, type checking, etc. These are all part of the modeling language and its development environment. A cornerstone of our approach is the correct by construction generation of multithreaded code after it has been verified during compilation. This code is then distributed automatically on a high integrity middleware that runs atop the underlying hardware. In the next section we will describe our specific implementation and the tools that were created to implement and test the feasibility of this methodology.

4.3 Toolchain

Further, through its *system* component's capabilities of modeling hybrid or generic components and the ability to define multiple implementations for a component type, AADL enables gradual refinement from initial requirements to executable models with the flexibility for rapid design implementation changes. Too seldom do these approaches address the issues of combining models and tools for the purpose of insuring consistency between the modeled specifications and the generated executable. Often, the distributed applications must be written manually, a tedious task even when using middleware as a buffer between differing underlying hardware platforms. We wish to integrate automated code generation tools for the formal language MRICDF with the base AADL tool set. By doing so we can insure correct by construction code generation from the model specification while reducing time to market.

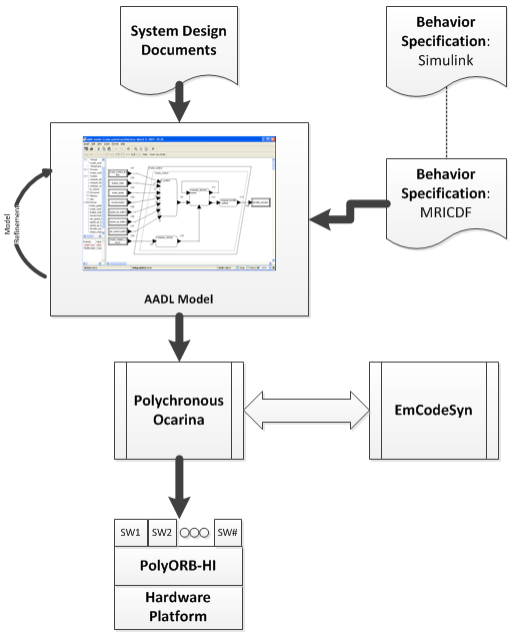

Figure 4.3: APECS Design Flow

Now we will discuss the APECS toolchain (illustrated in Fig. 4.3) and how it utilizes the formalisms and tools presented in the preceding section. In APECS the stakeholder provides their initial requirements in a form of their choosing: natural language, UML Use Cases, etc. The developers then must manually define the initial model in AADL using systems and generic devices to lay out the top level structure according to the initial specifications. The top level of the model is always a *system* component, typically referred to as the *Complete System*. This complete system is then refined by populating it and its subsystems with additional AADL components. Generally the execution platform is defined first, so that the software hierarchy may be properly structured and bound to the underlying hardware

as it is defined. As the refinement progresses more detail is added to the model by means of property definitions and use of the AADL language annex descriptions addition (non-) functional property details. Independently of this, the software processes are modelled in MRICDF. The MRICDF source code is then associated with its corresponding *process* component as shown in Fig. 4.4.

```
process MRICDF_App
   features
      ...
   ...
end MRICDF_App;

process implementation MRICDF_App.MA
   ...

   Properties
      source_name => "MRICDF_Source";
      source_language => "MRICDF";
      source_location => "../PATH";
end MRICDF_App.MA;
```

Figure 4.4: Associating MRICDF source files with AADL

After the AADL model has been created and the formal code bound to the software hierarchy, it is subjected to various forms of static analysis. Real-time schedulability, for instance, can be tested using Cheddar. Alternatively, different processor bindings may be tried to determine optimal distributions of the software components for the model's resources. Additional custom analysis techniques may be added by leveraging AADL's capacity for language extensions and including add-ons to the modeling environment that can interact with the custom annexes and properties. Once satisfied that the configuration and performance of the modelled system meets the requirements, the development proceeds to code generation. Ocarina is invoked on the AADL system specification. The new frontend of Ocarina (show in Fig. 4.5 has had two significant steps added before it passes the instance tree to the backend for code generation.

The MRICDF extension becomes active after the *instance tree* has been created. The tree is walked by the tool and any process component found has its properties searched for an MRICDF binding. Each such binding found identifies a process that must have its threads identified and generated before the code generation process can proceed. The first step in accomplishing this is to invoke another of APEC's tools - EmCodeSyn.

A library call is made to EmCodeSyn, invoking it on the source file(s) bound to the current

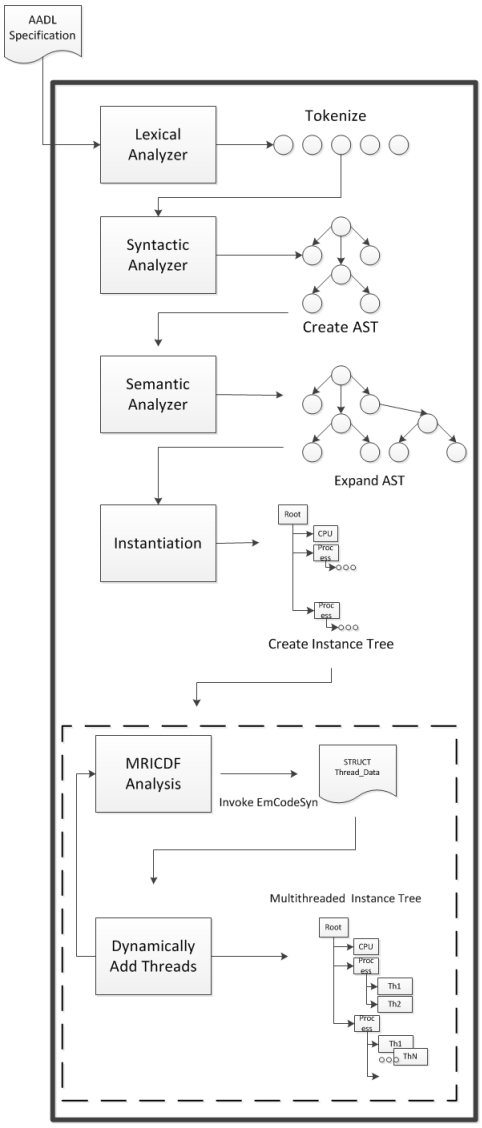

Figure 4.5: Ocarina Frontend with MRICDF extensions

target process. The MRICDF source files are analyzed and validated by EmCodeSyn before generating C/C++ code as described in [50], but with an exception. Normally, when EmCodeSyn generates C/C++ code from MRICDF it also automatically creates a *main* function. When generating code for APECS, EmCodeSyn instead only generates a library of the individual thread functions. The *main* with the necessary scheduling will instead be generated at the conclusion of Ocarina's backend code generation.

In addition to the location and name(s) of the file(s) to be analyzed, the call passes a pointer to a struct to EmCodeSyn. After completing its analysis, EmCodeSyn populates this struct, shown in Fig 4.6, with the number of threads found in the MRICDF source and the names

and types of any shared variables. Upon the return of this struct the next step in the

```
struct Data {
  Std::String Type;
  Std::String Name;
};

struct MRICDF_Threads {
  int NumThreads;
  Data* Shared_Variables;
};
```

Figure 4.6: Passing Thread Information

MRICDF extension begins, updating the current process based on the returned data (seen in Algorithm 1).

First a thread component type is instantiated, named "thread_{j}" (where j is the current thread number being instantiated). Then a thread component implementation, "thread_{j}.t" is instantiated. The new thread component implementation is added as a subcomponent to the current process's instance. To make use of the generated code, a subprogram is instantiated named "block_{j}". A subprogram call to "block_{j}" is registered to the corresponding thread instance. The necessary source properties are added the the subprogram to indicate the source function's name and where the encapsulating file is located. Additionally, the thread's dispatch protocol is specified by the addition of requisite properties to the thread's instance.

After the thread and its subcomponents and properties have been added, its also necessary to add any external shared data to the model. The second field of the struct is a two-dimensional array containing information on the interface and shared data of each of the threads. Each row represents a thread, and a column exists for each external or shared connection in the thread. The columns are a struct tuple containing two strings. The first string is the type of the variable, while the second string is the name. For each thread, its corresponding row in the variable array is traversed. The process is checked each time a variable is found to see if the data element already exists or if it is part of the process's interface. If it does not exist, then a new corresponding data component is instantiated using the type and name information from the array. Finally, data access permissions and connections are added to the process between the data element and the current thread. After the process has been populated with its threads, their subcomponents, connections, and properties the expanded instance tree is ready to be handed off to the backend for code generation.

Linking to the files containing these external functions, which have been generated from the MRICDF model by EmCodeSyn, is handled automatically by the generated Makefile. The full suite of generated code is organized and distributed as code libraries as shown in Fig.

Algorithm 1 Dynamically Adding Threads to an AADL instance tree

```
1: function ADD_MRICDF_THREADS(Thread_Struct)
2:     for all processes ∈ Tree do
3:         if Source_MRICDF ∈ process_i.properties() then
4:             for Thread_Struct → NumThreads do
5:                 new(type, "thread_{j}")
6:                 new(implementation, "thread_{j}.t")
7:                 process_i ← reg("thread_{j}.t")

8:                 new(type, "block_{j}")
9:                 new(implementation, "block_{j}.b")
10:                "thread_{j}.t" ← reg("block_{j}.b")

11:                "block_{j}.b" ← prop(Source_Loc)
12:                "block_{j}.b" ← prop(Source_Name)
13:                "thread_{j}.t" ← prop(Protocol)

14:                for all Thread_Struct → Vars do
15:                    if Vars_k → Name ∉ process_i then
16:                        if Vars_k → Type ∉ process_i then
17:                            new(type, "Vars_k → Type")
18:                        new(type, "Vars_k → Name")
19:                        process_i ← reg(Data.(Vars_k → Name))
20:                    Data_Access(Thread_{j}, Vars_k → Name)
```

4.7.

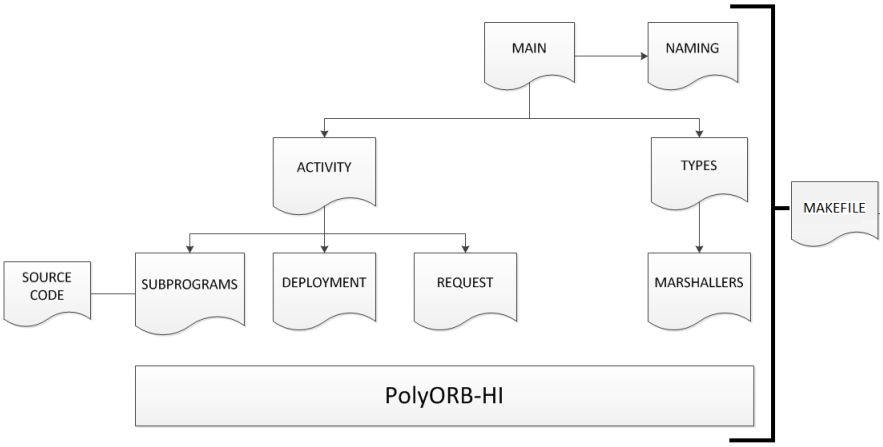

Figure 4.7: Code Distribution

Main: The main file initializes and starts the system tasks as defined from the AADL model and the linked source specifications.

Activity: Each task of the system is declared and defined in the Activity library. Tasks are derived from each of the thread components in the aadl model.

Subprograms: The Subprogram library defines the calling interface for each of the functions called by the subprogram calls of the tasks. Internally, they call an external function that is expected to be defined in the source files.

Deployment: The deployment library contains a protected object designed to act like a mutex for synchronizing task communication.

Request: The header of the library defines a struct that maps each port in the model. The struct contains two variables, the first is a union of structs called *var*. Each struct in *var* represents a port in the model, the content of these structs is a variable of a type matching that of the associated port. The second variable, *port*, identifies which port in the union is currently being targetted.

Marshallers: The marshaller library defines the functions to translate network stream data into application usable data.

PolyORB−HI: Elements from the PolyORB−HI middleware libraries are included as needed by the generated libraries.

Source Files: The code generated from invoking the EmCodeSyn tool on the linked MRICDF specifications.

The generated Makefile handles the compilation and linking of the generated and user source files automatically. In our prototype framework AADL provides the versatile and extensible foundation for the co-modeling of the system's execution platforms and and software hierarchy. The introduction of MRICDF as a behavioral specification supplies a formal, mathematically rigorous model. Its addition creates a more complete system description and allows for both automated multithreaded code generation and verification of the software. The framework's comprehensive capture of the system's structure allows for earlier analysis of the system and its modular design enables component reuse and refinement from an initial definition to a verified and fully refined implementable model. In the next chapter we will describe the process and tool implementation for our optional Simulink frontend extension for the APECS framework. The goal of this extension is to provide a frontend with industry familiarity and support the use of existing legacy code. By automating and verifying its translation into a formal intermediate format additional front

Chapter 5

Simulink to Polychrony

5.1 Introduction

Simulink is a simulation-based, model driven design tool using block diagrams to describe dynamical systems. Simulink models are organized as functional blocks representing piecewise-constant functions that operate on discrete-time signals. The primitive blocks of the model can be broadly grouped into two categories: *Interface* and *Combinatorial*. Interface blocks connect the model with its environment and fall into the subcategory of sources and sinks. These connections may be to an external source (e.g. *FromFile*, *FromWorkspace*), to another Simulink system (*Inport*, *Outport*), or to something internal (*Constant*). *Combinatorial Blocks* are those that modify the values of the signals as they move from source to sink, applying operations to one or more inputs and *instantaneously* producing one or more outputs. These operations may be arithmetic, relational, logical or something more advanced. In addition to these broad, basic categories there are some temporal blocks that modify the rates of communication and preserve the states of the system (e.g. *Zero-Order Hold*, *Unit Delay*). *Subsystems* are user-defined abstractions that encapsulate primitives and other subsystem blocks. They define an interface of input and output data ports that creates a standardized interface between the environment and the system's internal components. The internal components of default subsystems may inherit their sampling rate from their inputs or they may have their own explicitly defined timings. There also exist two variants of the system with differing implementations of control for the system's execution. The *triggered* subsystems receive an additional control signal. The system and its subcomponents only activate during instants when a triggering event occurs. A Triggering event is reflected by a change in value of the control signal either rising, falling or both depending on the user selected settings. Alternatively, an*enabled* subsystem operates during instants where the control input, 'enable', is set. Both of these subsystem variants affect their subcomponents beyond merely establishing hierarchy, complicating the timing requirements in ways that will be explored in more detail in section 5.2.

The ultimate goal of this translation is the facilitation of an automated toolchain where software behaviors specified in Simulink are transformed into MRICDF, verified, and then used EmCodeSyn's code generation capabilities to create executables targeted for a particular platform. In pursuit of this goal we attempt to develop a method of model transformation that, given the same inputs, will produce a new model with identical output behavior. This approach is supported by the similarities between MRICDF and Simulink models. Both approaches utilize a graphical approach to model data-flow networks. With Simulink organized into Subsystems and MRICDF grouped by composite actors both are organized hierarchically in definition as well as in execution. This means the overall structure of the model, if not the exact composition, can be preserved after transformation. However, despite the broad similarities between the two languages there are a number of key differences that make this translation worthwhile albeit complicated. At the core of this is the issue of formal semantics. While MRICDF's semantics are formal and precise, those of Simulink differ based on simulation settings and their intended behavior is only partially documented in natural language. To enable an accurate translation to a more formal representation it is necessary to enforce certain additional restrictions. For instance, while MRICDF is strongly typed, Simulink allows signals with implicit. MRICDF's polychronous model of computation is entirely built around a discrete model of time while Simulink operates in continuous time. In fact, even the blocks of Simulink's discrete library technically operate as piecewise continuous functions. While performing discrete time simulation, each block in the Simulink model either has an explicitly specified sampling period or it inherits a sampling rate from its sources. Because the system will be interacting with a larger environment of potentially unknown timing we require that all of the top-level inputs of the model have explicitly specified a sampling rate. While MRICDF operates in logical time, it is necessary to know the sampling rates of the system model to determine the blocks' computational rates relative to one another to ensure those constraints are enforced in the generated model and to statically determine whether the constraints present any conflicts.

We choose a *fixed step and discrete* simulation solver for the model. This solver computes the next time step at each instant by adding a fixed-step size increment to the current time as variable step-size is unsuited for models intended for code generation, it does map to real-time clocks. It also disallows the use of continuous blocks. At this time, we don't support continuous block behavior. The execution mode is left to *auto*, as either single or multiple tasking models are compatible with our approach. Auto will choose the mode appropriate to the current model. Single task mode executes all blocks together at every step while a multiple tasking groups blocks according to their execution rates as representative of concurrent, multi-rate executions. Finally, we assume that the *Boolean logic signal* (BLS) flag is set. This flag forces the output of logical operations to be *Boolean* typed. In addition to being good practice this also simplifies the type inference and verification process.

The translation of MRICDF to Simulink proceeds as shown in Fig. 5.1.

The Simulink model is converted from its original format (.mdl) into an XML file using the 'ExportToXML' feature of Simulink's *save_system* command. The XML source is then

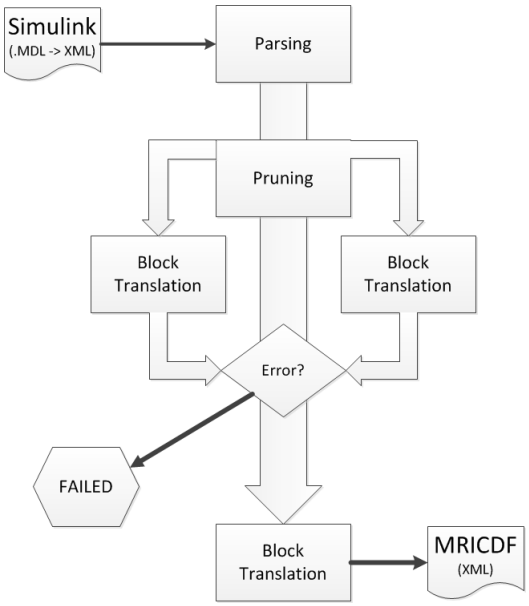

Figure 5.1: Sim2Em Translation Flow

parsed by our tool building a list of blocks and mapping their typed connections. As they are parsed, the block types are checked to make sure they are supported for translation. If a conflict is found then an *error* is generated along with a message identifying the unsupported block. Additionally, during this step some virtual blocks are pruned from the model before it undergoes static analysis. For example, the *Goto* and *From* blocks that are used to route signals in place of drawn lines. If found, a standard connection annotated with the type and port information from the *Goto* block is added to the model for each connected *From* block and the virtual blocks are deleted. The removal of these blocks is necessary because the symbolic connections made by these blocks is not reflected in the structure of the model which disrupts inferencing attempts that are made based on port interconnections. After the blocks of the model have been parsed, the port data types and activation rate information necessary for building the corresponding MRICDF network must be extracted. The applications of these properties and the methods of their extraction is discussed in detail in sections 5.2 and 5.3. Whether the model types or timings are extracted first does not matter, so long as both are completed before the translation process begins. If an error or other irreconcilable conflict is found in the model's properties during this extraction process then the translation will fail, citing either a type or clock related failure. Otherwise, the translation proceeds using the extracted properties to hierarchically construct the equivalent MRICDF actor network from the top down. Each basic block becomes an equivalent MRICDF actor. Systems and subsystems are translated in a top-down approach, becoming composite actors which are then populated by the translation of their component blocks.

5.2 Type Inference

Unlike MRICDF, Simulink does not require that each signal have an explicitly defined type. That is not to say that Simulink ignores the data types of its signals. Indeed its simulation engine enforces a set of typing rules and may even reject a model if these rules are found to have been violated. We present an informal overview of those rules here. First, Simulink assumes that a signal's type is *double*. There are two exceptions to this assumption. The first is where the designer has explicitly set a different type in the block's definition. The second is when the source or sink block of a connection requires a different type, because of a property of a block (e.g. logical operations require Boolean typed signals). Each blocks accepted port types are listed in their respective block specification found in [37]. Simulink will statically analyze a model using these assumptions and will reject any that attempt to connect ports with incompatible types. The basic types supported by Simulink are: double, single, signed and unsigned integers from 8 to 32 bit precision, Boolean, and undefined. Together this type set is denoted by the name T_{Sim}. For many blocks it is useful to refer to two type subsets: numeric ($T_{SimNum} = T_{Sim} - Boolean$) and *Boolean*. The compatibilities between each basic type is represented by a *type lattice* shown in Fig. 5.2.

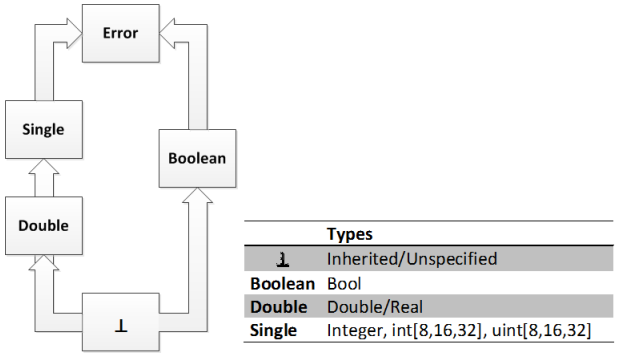

Figure 5.2: SIMULINK Type Lattice [66]

The existence of the undefined types, denoted by $\perp$ at the base of the lattice, is the reason the type inferencing step is required. In order to create the MRICDF network we must first obtain explicit types for all of the signals to be instantiated. In addition, some types may need to be refined using the lattice to avoid conflicts in the generated model. The type inference process is performed using a breadth first traversal of the model (See Algorithm 2). The fundamentals of this process are based on the type inference approach used in [49, 66] as the typing needs of MRICDF are the same as those of Lustre and Signal. Although the algorithm undertaken to achieve this inferencing is distinct.

At the beginning of the traversal the inputs at the root system level as well as other independent source blocks, such as *constants*, are pushed into a dequeue. Since these inputs are at the top level of the model, their sources are in the environment or in other systems

Table 5.1: Types of Supported Simulink Blocks [49]

Block	Type
$Constant_\alpha$:	$\alpha, \alpha \in T_{Num}$
In, Out Port	$\alpha \rightarrow \alpha, \alpha \in T_{all}$
Zero-Order, Hold, Unit Delay :	$\alpha \rightarrow \alpha, \alpha \in T_{all}$
Gain :	$\alpha \rightarrow \alpha, \alpha \in T_{Num}$
Arithmetic :	$\alpha \times ... \times \alpha \rightarrow \alpha, \alpha \in T_{Num}$
Trigonometric :	$\alpha \times ... \times \alpha \rightarrow \alpha, \alpha \in T_{Num}$
Relational Operator :	$\alpha \times \alpha \rightarrow Boolean, \alpha \in T_{Num}$
Logical Operator :	$Boolean \times ... \times Boolean \rightarrow Boolean$
Saturation :	$\alpha \rightarrow \alpha, \alpha \in T_{Num}$
Difference :	$\alpha \rightarrow \alpha, \alpha \in T_{Num}$
Switch :	$\alpha \times \beta \times \alpha \rightarrow \alpha, \alpha \times \beta \in T_{Num}$
Discrete Filter :	$double \rightarrow double$
Pulse Generator :	α, α
Data Type Conversion :	$\beta \rightarrow \alpha, \alpha, \beta \in T_{all}$
Digital Clock :	$\alpha, \alpha \in T_{Num}$
Subsystem :	$special^*$
SFunction :	$special^*$

* The type behavior of these blocks is described in the algorithm description

Table 5.2: SIMULINK Block Equations [66]

Equation	Type Equation
$y = Constant_\alpha$:	$y^T =$ if $y^T \leq \alpha$ then α else error
$y = FromWorkspace$	*
$y = Inport_\alpha, y = Outport_\alpha$	$y^T = x^T$
$ToWorkspace$	$y^T = x^T$
$y = Zero - OrderHold(x)$:	$x^T = y^T$
$y = UnitDelay(x)$:	$x^T = y^T$
$y = Gain$	$x^T = y^T = sup(double, x^T)$
$y = Arithmetic(x_1, \ldots, x_n)$:	$y^T = {x^T}_1 = \ldots = {x^T}_k = sup(double, y^T, {x^T}_1, \ldots, {x^T}_k)$
$y = Trigonometric_\alpha$	$y^T = x^T = sup(double, x^T)$
$y = Relational(x_1, x_2)$:	${x^T}_1 = {x^T}_2 = sup(double, {x^T}_1, {x^T}_2)$, $y^T = Boolean$
$y = Logical(x_1, \ldots, x_k)$:	$y^T = {x^T}_1 = \ldots = {x^T}_k = Boolean$
$y = Saturation$	$y^T = x^T = sup(double, x^T)$
$y = Difference$	$y^T = x^T = sup(double, x^T)$
$y = Switch(x_1, x_2, x_3$:	${x^T}_1 = {x^T}_3 = y^T = sup({x^T}_1, {x^T}_3, y^T)$
$y = DiscreteFilter_\alpha$:	$y^T = x^T = if y^T \leq$ double then double else error
$y = PulseGenerator_\alpha$	$y^T =$ if $y^T \leq \alpha$ then α else error
$y = DataTypeConverter_\alpha(x)$:	$y^T = if y^T \leq \alpha$ then α else error
$y = DigitalClock$	$y^T = double$
$y = SFunction$	*
$y = Subsystem$	*

* The evaluation of the types of these blocks is described in the algorithm description

outside of the model. Therefore if their type isn't explicitly defined in the model then it must be assumed to be *double*. In the case of *constant* blocks, the type must match that of the block's value parameter, otherwise the the model is rejected with a type error violation. After each block is evaluated, any changes in type are propagated to its connections. Any block not already in the dequeue that is updated through such a connection is pushed into the dequeue for evaluation. Each subsequent block is evaluated according to its corresponding typing rules and equations as described in Tables 5.1 and 5.2. The $\leq$ operator denotes a relational order between types on the type lattice. For instance $int32 \leq double$ while $Boolean \not\leq single$. The *sup()* operation on the other hand defines the greatest common refined type between two or more types on the lattice. For example, *sup(double, int32) = int32* while *sup(int32, Boolean) = Error*.

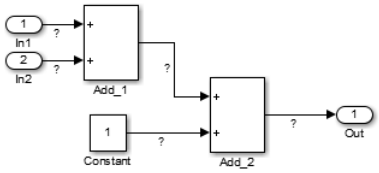

Figure 5.3: Arithmetic Typing Example

In the arithmetic operation modeled in Fig. 5.3. the two inputs and the constant are evaluated first. Since the inputs are connected to the environment and their type is undefined they are assumed to be *double*. After the two inputs have been processed, the first *add* block is appended to the dequeue for evaluation. As an arithmetic block, its supported types belong strictly the the numeric subset of type primitives. Referring to the type equation table, the type of an addition block is the *Sup()* of its inputs and its output and a *double* - which forces an *Error* result for non-numeric values. In this case:

$$\begin{aligned} & Sup(double, y^T, x^T{}_1, x^T{}_2) \\ \Rightarrow\ & Sup(double, \perp, double, double) \\ & \Rightarrow double. \end{aligned}$$

The *constant* contains an integer with the parameter value '1'. The the second *add* blocks type equation takes the form:

$$Sup(double, \perp, double, int32) \Rightarrow int32.$$

The result is this propagates to the addition block's output as well as back to the first adder, changing its output type requirement to *int32* causing it to be pushed into the front of the dequeue to be reevaluated. Reapplying the *Sup()* operator with the updated data types cause its inputs to also change to *int32*. This further back propagates to the inputs. If the designer had wished to first carry out an addition operation using double precision and then use an integer increment, they could achieve this by explicitly carrying out static

type casting using a *Data Type Converter* block. This repeated evaluation and and type propagation process continues until all the blocks have been evaluated with no pending type changes remaining. Two special cases for the listed type equations are the *Subsystems* and *S-Functions*. Each of these may have input and output types in any combination required by the designer. In the case of subsystems, the block hierarchy is flattened so that the subsystem's component blocks are analyzed directly. Once the types of its input and output subcomponents have been determined, those types can be compared with the subsystem's external connections. SFunction internals, however, are a black box from the perspective of the Simulink XML schema. As a result the tool is unable to directly validate the types of the SFunction ports. One approach to dealing with this is to assume the types of the interface have been explicitly set and verified by the designer. However, this is not ideal for safety and security validation of the system. It also causes issues for extrapolating the type interface of the MRICDF function. An alternative approach, illustrated in the PMU case study, is to create specialized subsystems that define the type interface of the SFunction. This allows the block's to be automatically extracted and applied to the generated function.

5.3 Clock Inference

As discussed in section 3.1.2, time in MRICDF is measured in discrete, logical instants. Each signal s has a clock b_s associated with it that represents a totally ordered sequence of Booleans. A value of *true* in b_s means that s is present and has a value in that instant. Unlike with monochronous languages such as Lustre or Esterel, these clocks are not necessarily related to a global base clock. Because they needn't all synchronize, the logical instants of the system of the system are said to be partially ordered. The temporal relationship between signals may be explicitly defined using clock calculus annotations[40] or they may be inferred from the properties of the actors.

Simulink simulates the behavior of its blocks using piecewise-constant continuous-time signals that are updated at intervals defined by ticks of a *global simulation clock* (T_n). The period of the global clock is defined for the model as a measure of real-time in terms of seconds or fractions of seconds. Each block in the model then has a *sample rate* that is defined by a period (π) and an initial phase (θ) such that when $T_n = N * \pi + \theta$ the block samples its inputs and updates its output. The rate of a signal is determined the block producing it and may be scaled to interact with blocks operating at greater or lesser rates through the use of *Unit Delay* and *Zero-Order Hold* blocks respectively. A special case for Simulink timing is that of *triggered* and *enabled* systems. Unlike a basic block, subsystems may optionally have an additional control signal that determines whether it is active. A *trigger* activates when the control signal crosses zero and may be either *rising*, *falling*, or *either*. A subsystem is *enabled* so long as the signal driving the *enable port* holds a value that is greater than zero. Because these components are disabled at other times the subcomponents of *triggered* subsystems must have sample rates that match that of the trigger, while those in an *enabled*

Algorithm 2 Type Inference

```
function INFERENCE(BlockList)
    Change_In, Change_Out, TypeDequeue, Fail
    for all block_i ∈ BlockList do
        if (block_i.type == 'Input' and block_i.parent == 'Root') or block_i.type == 'Constant' then
            TypeDequeue.push_back(block_i);
            BlockList.remove(block_i);
    while ! TypeDequeue.Empty() do Fail = TypeEval(block_j);
        if Fail then
            Throw Error
        else
            TypeDequeue.pop_front()
            UpdateSources(block_j, &TypeDequeue)
            UpdateSinks(block_j, &TypeDequeue)
function UPDATESOURCES(Block, Dequeue)
    for all Block_j.sources do
        if OutPort.type != block_j.InType then
            OutPort.type = block_j.InType
            TypeDequeue.push_front(source_k)
function UPDATESINKS(Block, Dequeue)
    for all block_j.Sinks do
        if InPort.type != block_j.OutType then
            InPort.type = block_j.OutType
        if Sink_k ∉ TypeDequeue then
            TypeDequeue.push_back(Sink_k)
```

subsystem execute in the subset of instants where their normal activation coincides with their parent subsystem being active.

As previously mentioned, each block has a specified sampling rate. By default this rate is -1 which signifies that its rate is inherited from its inputs. It is therefore necessary to infer the rates of these blocks to insure the communication between actors in the MRICDF network can be properly synchronized. Similar to the type inference extraction, our approach shares the same temporal concerns as the other synchronous and polychronous translations carried out in [49,66]. The fundamental requirements are the same for our MRICDF translation, and will be repeated here. However, the approach to extraction and its subsequent utilization differ, particularly with regard to the timing of conditionally activated subsystems. While the sampling rates are inferred the blocks are also checked for incompatibilities between their rates and those of their inputs. As with Type Inferencing, the clock inferencing process is carried out breadth first. It begins with the inputs and other independent source blocks at the root system level. Because these blocks either have inputs that are part of the external environment or they simply have no inputs at all it is assumed that they have explicit non-inherited rates. If not then the analysis will fail and throw a clock inferencing error. Otherwise the inference traversal continues, comparing each connected pair of blocks. The result of this comparison has several distinct possible outcomes. If the block is a *triggered* subsystem the timing of each input must be equal and must match the rate of the triggering signal. If this is not the case then the inputs will not be able to be synchronized with the subsystem's periods of activity and data will be lost. For other blocks, if the rate is specified explicitly then it is compared with those of its inputs. Their rates must align such that there exists an integer coefficient k such that for each input $k * R_{in} = R_{block}$ or $R_{in} = k * R_{block}$ where R_{in} and R_{block} are the rates of the inputs and the block respectively. Finally, if the rate is inherited then the inferred rate should be the greatest common divisor (GCD) of the rates of its inputs. This can be somewhat complicated by the presence of differing phases in the input rates. The way the sampling rate (π, θ) is calculated is given by the following formulae:

$$\pi = \begin{cases} gcd(\pi_1, \pi_2) & \text{if } \theta_1 = \theta_2 \\ gcd(\pi_1, \pi_2, \theta_1, \theta_2) & \text{if } \theta_1 \neq \theta_2 \end{cases}$$

$$\theta = \begin{cases} \theta_1 & \text{if } \theta_1 = \theta_2 \\ 0 & \text{if } \theta_1 \neq \theta_2 \end{cases}$$

For example, assume we have an addition block α that receives two inputs with the rates (6,2) and (4,2). Then α's inferred rate is :

$$((6,3),(4,3)) \Rightarrow (2,1)$$

Similarly, if the inputs had the rates ((2,1), (5,3)) then α's new inferred rate would be:

$$((2,1),(5,3)) \Rightarrow (1,1)$$

By lowering the period of the sink block, we insure that its possible for the sink block to be active at any instant where an input may occur. During block translation we will apply

these inferred properties to synchronize communications between blocks. The pseudocode for our algorithm is displayed in Algorithm 3. A block type of *Source* if its part of the Simulink *source* library and is at the *Root* level of the model, therefore either has no inputs or is receiving its inputs from the environment. The internal input sources of a subsystem will return the external block connected to the corresponding input port of that subsystem.

Algorithm 3 Clock Inference

```
function INFERENCE(BlockList)
    Change_In, Change_Out, TypeDequeue, Fail
    for all block_i ∈ BlockList do
        if (block_i.type == 'Source') AND block_i.rate == -1 then
            return ERROR
        else if (block_i.type == 'Triggered') then
            if block_i.rate == -1 then
                block_i.rate = GCD(block_i.sources)
            if ! Match(InputRates(block_i)) then
                return Error;
        else if ((block_i.type != Enabled) OR (block_i.type != Subsystem)) then
            if block_i.rate == -1 then
                block_i.rate = GCD(block_i.sources)
            else
                if ! MultOrDiv(block_i) then
                    return Error
```

At this point each port and associated signal has a known type and rate associated with it. If no errors have been found then the system may progress to *block translation*. In the next section we will describe the block translation process and show how the extracted properties are applied to the generated actor network. Then we will show how these extracted properties are applied to the translated model.

5.4 Block Translation

Having inferred and validated the types and timing properties of the Simulink model, the actual translation may be performed. Due to the discrete nature of MRICDF we have limited the translation to elements of Simulink's discrete library. These supported blocks also include mathematical operations and select virtual blocks. The complete listing of the currently implemented block translations is given in Table 5.3.

The structure of the Simulink model is such that the component blocks and subsystems of a (sub)system are nested within their parent block's definition. This arborescent arrangement is similar to the hierarchical structure of MRICDF. A top-down approach is therefore

Table 5.3: Supported Simulink Blocks

Block Category	Supported Blocks
Temporal	Difference, Unit Delay, Zero-Order Hold
Operational	Arithmetic, Relational, Logical, Trigonometric, Gain
Sources and Sinks	Pulse Generator, Input, Output, Scope*, Constant, Digital Clock
Typing	Data Type Conversion
Routing	From*, Goto*, Switch
Filters	Saturation, Discrete Filter
User Code	S-Function
Organization	Subsystem, Triggered Subsystem, Enabled Subsystem

* The evaluation of the types of these blocks is described in the algorithm description

employed to maintain the organization of the source model. Starting from the root system, we translate each block by first constructing its interface and then populating it with any internal subcomponents before finalizing its internal connections.

Simple and Complex Blocks

For simple Simulink blocks this process is fairly straightforward, as most have a direct counterpart in MRICDF. A *unit delay* block, for instance, becomes a *buffer* actor with the same initial values. By the same token, basic operators become *function* actors with the same interface parameters and a body defined in C code that performs the same operation. For example, a translation of an addition operator from Simulink to MRICDF is shown in Fig. 5.4. More complex blocks, particularly operations that maintain some internal state, become *composite* actors. For example, the *difference* block outputs the difference between the current input and the input of the previous computational instant which requires that the previous input value be preserved for the next instant. The translation of this block is shown in Fig. 5.5. The simulink difference block is shown on the left. It is translated into the composite actor "diff" which has a single input and a single output. The composite contains four actors: an input and an output actor that correspond to the interface ports, a *buffer* actor that preserves the previous input value, and a *function* actor that performs the subtraction of the previous value from the current input.

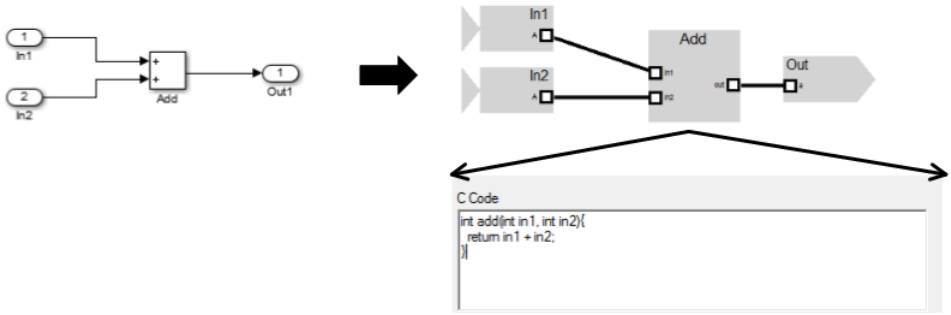

Figure 5.4: MRICDF Adder

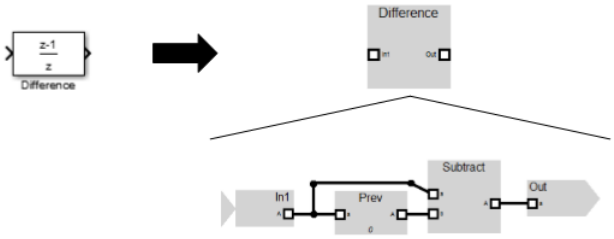

Figure 5.5: MRICDF Difference

Subsystems

The vast majority of practical applications will be composed of many nested subsystems. These subsystems may be conditionally or unconditionally activated. In either case the initial approach is the same, the system interface is instantiated as a composite actor and then recursively populated with an actor network representing its subcomponents. if the subsystem is unconditional then its translation is complete at this point. If, however, the subsystem is classified as either a *triggered* or *enabled* subsystem then additional steps must be taken to constrain the activation of these actors. The *triggered* subsystem is a system with an additional *trigger* port. The trigger may be activated by either a rising or falling edge or both. An edge is defined as a change from negative to positive or positive to negative in the signal. During instants where no edge occurred, the system is inactive. An inactive triggered system holds its output values and preserves any discrete internal states until its next activation.

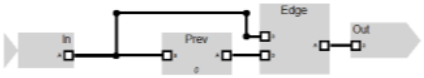

Figure 5.6: Trigger Port Translation

Internal states are preserved in buffers during periods of inactivity. Similarly, outputs are latched in buffers and repeat their previous value during instants of inactivity.

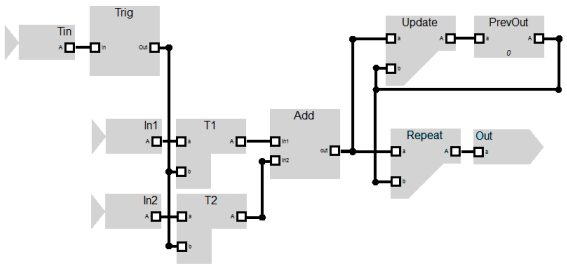

Figure 5.7: Inserting Triggers into the System

Similar to *triggered* subsytems, *enabled* subsystems have an additional *enable* port that controls when it is active. Rather than being edge triggered, this variant is active so long as the enable signal is positive. An enabling or disabling event is generated when *zero crossing detecting* finds the control signals value has changed. Unlike a triggered subsystem, which

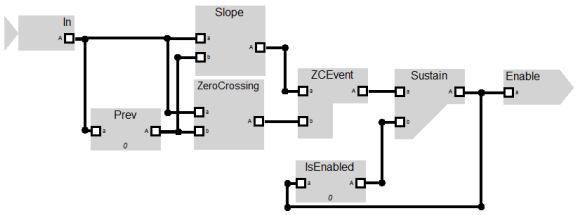

Figure 5.8: Enable Port Translation

always maintains its internal state during periods of inactivity, the enabled subsystem may opt to reset when activated and optionally it may hold or zero its output while inactive. We currently default to the same inactive behavior as the triggered subsystem and preserve the states and outputs between periods of activity. Also, unlike the triggered variant, the enabled subsystem's components run normally at their defined rates while the block is active rather than being wholly restricted by the rate of the enabling signal.

Special Cases

Some supported blocks are considered a special case, in that they are not translated directly into the MRICDF model. For example, the previously mentioned virtual blocks *goto* and *from* symbolically route the signal from Goto's input to From's output without drawing a connection. MRICDF does not provide such symbolic routing but the issue is easily resolved by replacing goto/from pairs with an equivalent literal connection as shown in Fig. 5.9. Similarly, the *scope* actor that Simulink employs to track values during simulation would not translate directly to MRICDF as EmCodeSyn is not a simulation environment. While the final deliverable would reasonably be expected to have such debugging components removed, its a commonly used technique during the model's development. Therefore to

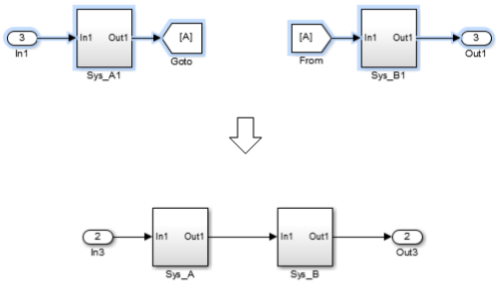

Figure 5.9: Virtual Connections

better enable a quick translation flow, scopes are supported as an output to file block in the generated MRICDF model. This allows for greater compatibility across development phases and provides an execution trace of the scoped signal for analysis if desired.

Applying Inferred Knowledge

Once the blocks have been translated into actors, we employ the properties that were inferred in the previous steps. Each actor's ports and connected signals are assigned the type of their Simulink counterpart. Although, since MRICDF only natively supports the types *integer*, *real*, and *Boolean* the types from Simulink are mapped to an equivalent MRICDF type as shown in table 5.4

Table 5.4: SIMULINK to MRICDF Type Mapping

Equation	Type Equation
Boolean	bool
single, double	real
int16, uint16, int32, uint32	int

After the blocks are defined, the connections between them are created. However, before this can be done we need to check the system for rate conflicts. For each potential connection, the source and sink blocks are checked with three distinct possible outcomes. The first is that the blocks have identical rates in which case no additional changes are required. The second case is that the source block is slower than that of the sink block. In such a situation a new composite actor is added that will *oversample* the source signal. Recall from the clock inference step that the relative rates of connected signals are required to be integer multiples. Thus $\frac{\pi_{sink}}{\pi_{source}} = k$ where k is an integer ¿ 1. The oversample actor samples the source signal and generates k successive copies on its output. The composition of the Oversample actor is

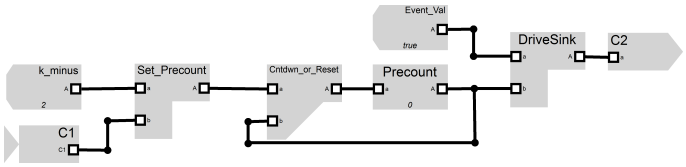

Figure 5.10: Oversampling Composite

shown in Fig. 5.10. When an input occurs the count is reset to the constant $k-1$ and the input is propagated through the priority input of the Merge_Val actor to the output. the count then decrements by 1 each instant, outputting a copy of the most recent input from the *prev* buffer until the count reaches zero. We know that a new input will arrive every k instants so we explicitly set a clock relation between the count such that $in^{=}when(cnt == 0)$. Fig 5.5 gives a sample trace of a sink actor that is twice as fast as its source.

Table 5.5: Overclock Trace (K = 2)

t	0	1	2	3
In	In_1	$\perp$	In_2	$\perp$
Cnt	0	1	0	1
M	1	0	1	0
Out	In_1	In_1	In_2	In_2

Because the rates of communicating actors were confirmed as multiples during the initial extraction and analysis phase we do not need to be concerned with the phase of the communicating blocks when calculating the count for the oversampler. The reason for this is because in order to be considered a multiple, the phase of the sink actor must either be equal to the phase of the source actor or it must be zero while the phase of the source must be an integer multiple of the sink block's period. In either case the actor is inactive until the first input arrives, after which point the phase of both has already occurred.

The third possibility is that the source is faster than the sink, in which case a composite actor is inserted to *undersample* the source signal. This process is similar to that used by the oversampler except instead of generating k copies of each input it filters the signal to allow only every K^{th} value through. The structure of the undersampler can be seen in Fig. 5.11. This actor fires every time it receives an input, thus we explicitly add the rate relation $In^{=}cnt$ to reflect the fact that the count buffer should be sampled at the same rate as the input. The initial value of cnt, associated with the first input instant, is set to one. This allows the initial input to be passed through to the output. In the next instant count will be zero which will trigger the sampler $S1$ to reset the count to the constant value $k-1$. Each subsequent instance decrements the count until it again reaches zero. Fig 5.6 gives a sample trace of a source that has a $k = \frac{\pi_{source}}{\pi_{sink}} = 3$.

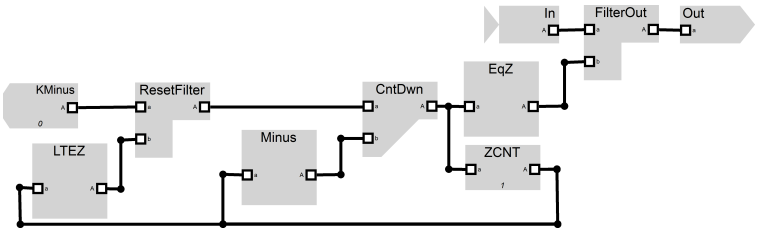

Figure 5.11: Undersampling Composite

Table 5.6: Underclock Trace (K = 3)

t	0	1	2	3	4
In	In_1	In_2	In_3	In_4	In_5
Cnt	1	0	2	1	0
M	0	2	1	0	2
Out	In_1	$\perp$	$\perp$	In_4	$\perp$

As with the oversample actor we need not worry about the phase of the actors when calculating the count. Because Simulink forbids the use of a sampling rate where $\theta \geq \pi_2$ and because in order to qualify as a multiple $\theta_1 == \theta_2$ or $\theta_2 = 0$ and $\exists k, \theta_1 = k * \pi_2$ the sample phases for an undersampling relation must be equal. Otherwise $\theta_1 \geq \pi_2 > \pi_1$ which would violate Simulink's timing requirements. If the phases are equal then they are accounted for by the time of the arrival of the first input.

Chapter 6

Case Studies

In this section we will present case studies that illustrate the APECS process and results.

6.1 Elevator

The first case study models the *elevator system* of a five story building. Each floor of the building is serviced by a set of four *elevator cars*. A *call panel* on each floor places a request for service. The service calls and subsequent scheduling of the *elevator cars* are handled by a *central controller*

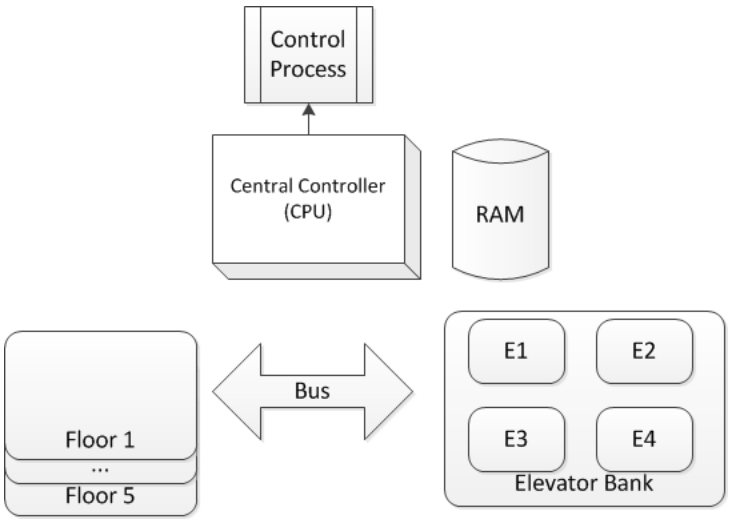

Figure 6.1: Elevator Top Level AADL Model

The *Complete System* of the Elevator (Fig. 6.1) is composed of a number of sub-systems and a central controller. For the sake of readability we only give a graphical representation of

the component composition the AADL systems and eschew representing the communication port connections. There are five instances of floor systems, once for each floor of the target building. An *Elevator Bank* subsystem contains the four elevator cars that service the building. Finally, its *central controller* that is made up of a CPU and its attendent memory and operating software. The subcomponents communicate through a shared bus.

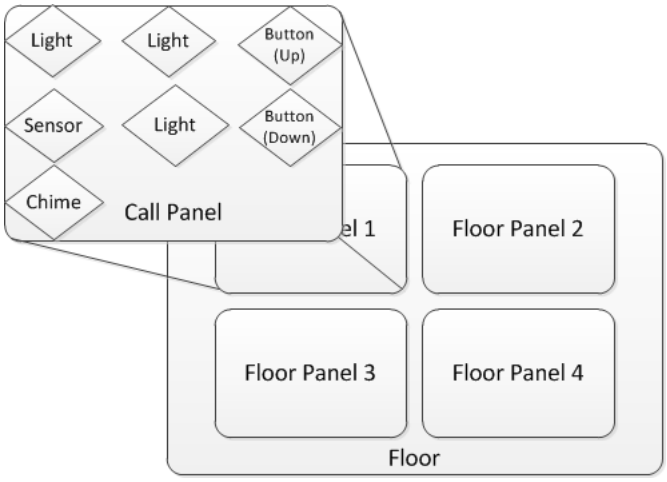

Figure 6.2: Floor Call Panel

The *call panel*s are subsystems of the *floor*s (Fig. 6.2). Each floor has four *call panels* that communicate via bus with the *central controller* to schedule *elevator car* service. The panel has two button *devices*, the first requests to go up while the second indicates a down request. There is a sensor *device* that detects when a car has arrived on that floor, and indicates it by deactivating the pending light associated with that request and sounding a chime.

The *elevator car* subsystem is composed of a *button panel* for user interfacing, a *controller* to process user requests as well as feedback from the other subsystems, a *car motor* that raises or lowers the car, and a *Door System* that opens and closes the car doors.

The *Door System* (Fig. 6.4)is responsible for opening and controlling the door of an *Elevator Car* as requested. It has its own microcontroller setup to process external inputs both from the environment and received from user requests via the car's *button panel*. In addition, it has a motor *device* that physically drives the opening and closing of the door. A timer *device* begins counting down after the door has opened, when it reaches zero the door will automatically begin closing unless some external intervention occurs. Lastly, a sensor *device* detects when there is an obstruction in the path of the door.

The first step toward creating the system model in APECS is to define the underlying platform and architecture. This is accomplished through a top-down approach and the application of iterative refinement. First, the top-level system, *Complete*, is defined. As this component encapsulates the entire model it generally has no defined *features*. Instead it serves as an abstraction that organizes the model. This isn't to say that it cannont

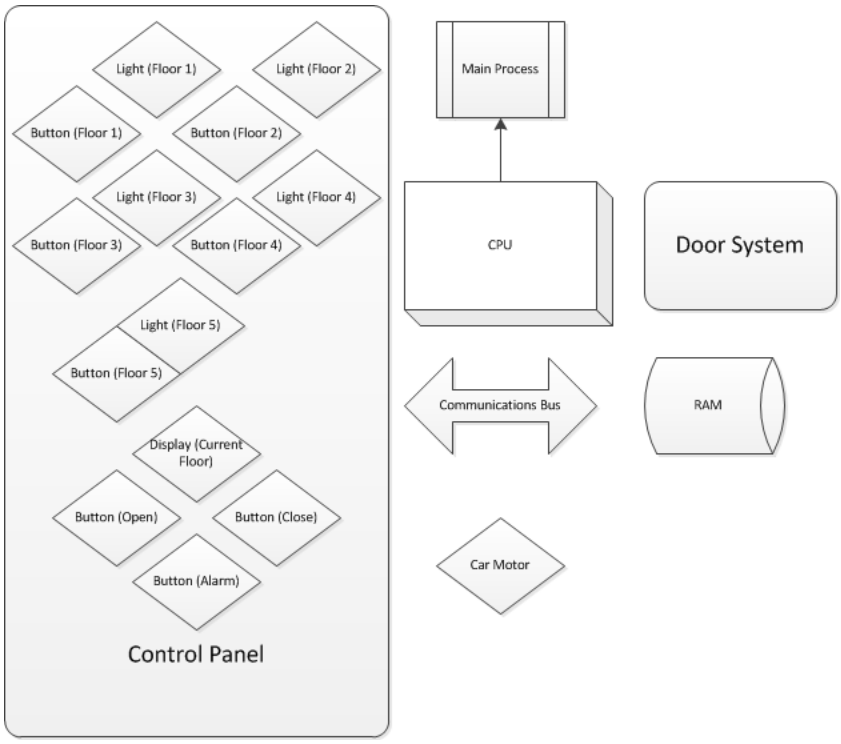

Figure 6.3: Elevator Car Control AADL Model

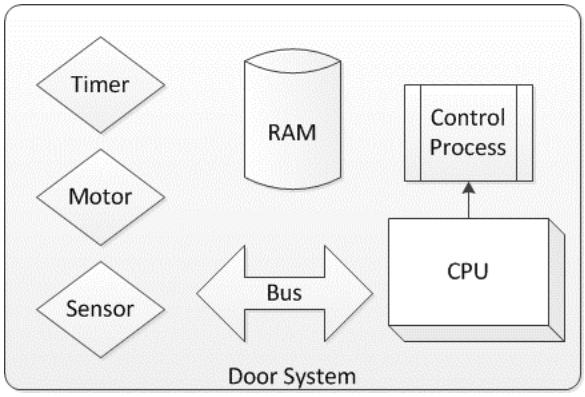

Figure 6.4: Door System AADL Model

interact with its environment. Even without ports it may still act or be acted upon through sensor or actuator devices or through bus communications. The contents of the elevator's top-level system are shown in Fig. 6.1. The microcontroller can be modeled as simply a processor component or it can be further detailed as it is here with its attendent memory, bus communication, and operating software. Each subsystem is defined in turn, with a standardized interface. The internal composition and flow of these components can be refined throughout the design process. The floor subsystem (Fig. 6.2), which is declared once and

instantiated as needed for each floor in the target building, contains the system for the call panels that serve as the interface for the passengers. The elevator car (Fig. 6.3) contains multiple subsystems, such as a control panel for user interaction and a controller that operates the door of the car. By virtue of its concise control system, multithreading potential, and safety criticality, the *Door System* will be the focus of this case study example.

Event	Conditions	Outputs
E1	OpenRequest	Motor <= open and Timer <= start
E2	Obstruction	Motor <= open and Timer <= start
E3	CloseRequest	Motor <= close
E4	Timeout	Motor <= close

Figure 6.5: Door State Table

After determining the top-down composition of the underlying execution platform in AADL, the software behavior specifications can be associated with their components in the software heirarchy. We start by specifying the desired behavior in natural language. The desired behavior of our elevator door, therefore, is:

- If an open request is received via bus from the *Elevator Car* parent system, then the door should signal the *motor* to begin opening the door and the *timer* to begin counting down until the door should automatically close. (The *Elevator Car* will screen out open requests if the car is moving.)
- When *Timer* finishes counting down, it signals the door's *central controller* that it should begin closing the door.
- If a close request is received via bus from the *Elevator Car* parent system, then the door should signal the *motor* to begin closing the door.
- If the *sensor* detects and obstruction at any point while closing, it signals the *central controller* to reopen the door and reset the timer.

The described behavior is depicted both in the state table (Fig. 6.5) and as a stateflow diagram in 6.6.

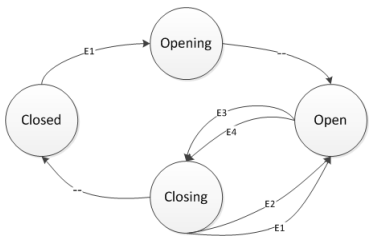

Figure 6.6: Door Controller State Flow

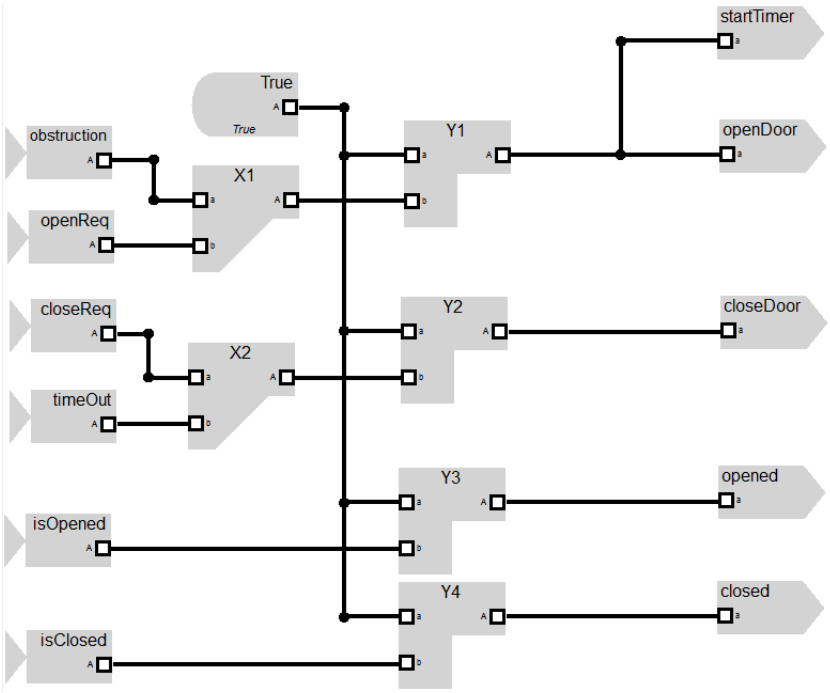

Figure 6.7: EmCodeSyn Model of the Door Control Software

We took this desired behavior and modelled it in MRICDF (Fig. 6.7).

This system receives six inputs. From the top, the first is from the *sensor* indicating whether an obstruction has been found. The next two are open and close requests from passengers relayed from the *elevator car*. Then timeout is relayed from the *timer* after the door has been open for the maximum alloted time. Finally, the last two input signals, come from the *motor* when the door has been either fully opened or fully closed respectively. The last two are necessary to update the *elevator car* with the current status of the car. Obstruction and OpenRequests both go to a *merge* actor, with obstruction taking the priority slot so it won't be blocked by an errant 'false' input via OpenRequests. The output of that *merge* actor drive a *sampler*'s selector input. While either an obstruction or OpenRequest are true a 'true' constant signal will be able to propagate through the sampler to activate the openDoor and StartTimer outputs. These outputs connect to the *motor* and *timer* respectively. Similarly, the CloseRequests and TimeOut inputs drive their merge actor, which in turn drives the output of a sampler that connects to *motor*'s CloseDoor signal. The last two signals, isClosed and isOpened drive *selector* actors that output true once the *motor* signals the door has been completely closed or opened. The outputs 'opened' and 'closed' are relayed to the parent system *Elevator Car* to notify its central controller of the current door status.

The AADL process that is modelled by the aforementioned MRICDF specification is given in Fig. 6.8. In this case, the data, thread, and subprogram subcomponents will all be generated automatically and added dynamically by APECS.

```
process Controller
    ...     // Define Interface
end Controller;

process implementation Controller.C

  properties
    Source_Language => MRICDF;
    Source_Name => DoorControl.xml;
    Source_Location => ..\\DoorControl.xml;
end Control.C;
```

Figure 6.8: MRICDF Process Declaration

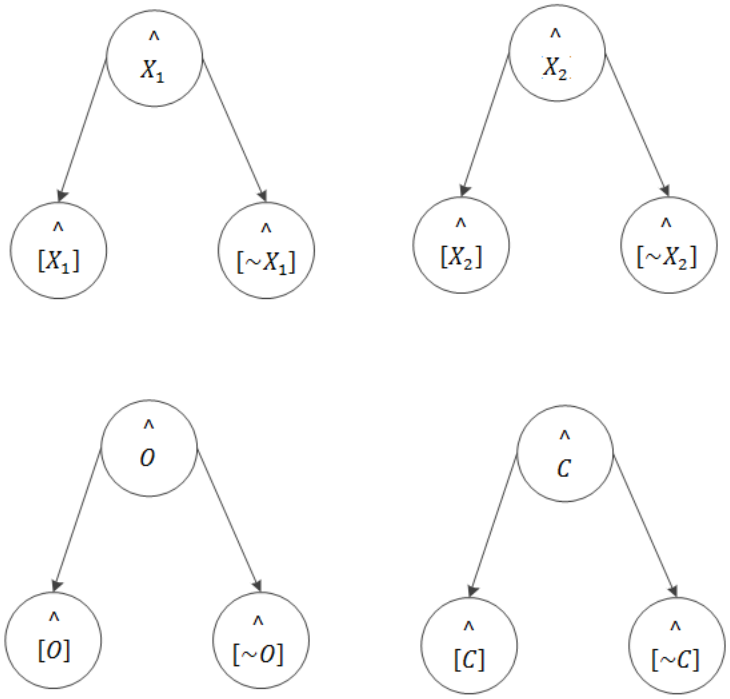

Figure 6.9: MRICDF Clock Tree

When APECS notices the MRICDF specification attached to the *process* definition, the

specification undergoes epoch analysis by EmCodeSyn. For the door system modelled above EmCodeSyn finds a set of partial triggers rather than a single master trigger. The forest of clocks generated is depicted in Fig. 6.9. These partial triggers mean that the control software can be divided into four indepedent threads, shown in Fig. 6.10. Thread 1 samples OpenRequest and Obstructions to control the open signal while thread 2 samples timeout and closeRequest singals to control closing the door. Threads 3 & 4 sample and update the parent systems control process on the current door status.

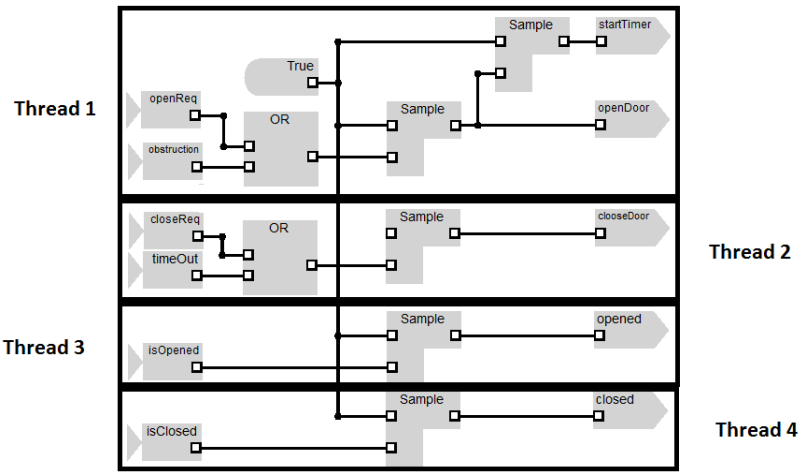

Figure 6.10: Resulting thread groups

These four threads are added as subcomponents of the of *Controller.C* in the *instance tree* along with the additional subcomponents as described in **Algorithm** 1. The updated model is handed off to the *backend* for code generation. A sample of the generated thread task for opening the elevator door is presented in Fig. 6.11 and the code generated for its subprogram call is presented in Fig. 6.12.

Fig. 6.11 shows a snippet of the code generated for the thread that singlas the door to open. The thread initializes itself and computes its activation period, based on the timing properties defined in the software architecture model, through the use of the middleware functionality. Each iteration of the loop, the thread calls the functions in its subprogram call sequence and then waits until its next activation period.

The function, "doorsys_block1" serves as a wrapper function that called block1(), which is declared in the generated code as an external function.

6.2 Phasor Measurement Unit

In this section we will describe the translation of an open-source Simulink model for a Phasor Measurement Unit (PMU) described by [58] for public use. A PMU provides measurements

```
void* openthread_job (void)
{


/*!
 * Waiting for other tasks initialization
 */
  __po_hi_wait_initialization ();
  __po_hi_compute_next_period
    (controller_openthread_k);
  while (1)
  {
    /*  Call implementation*/
    doorsys__block1 ();
    __po_hi_wait_for_next_period
      (controller_openthread_k);
  }
}
```

Figure 6.11: Generated OpenDoor Thread

```
void block1 (void);
void doorsys__block1 (void)
{

  block1 ();
}
```

Figure 6.12: Generated OpenDoor Function Call

of the magnitude and phase angle of AC voltages. *Synchrophasors* are a variant of PMU that provide time and location stamps along with the collected measurements. These synchrophasors are an essential tool for tracking and reporting on the health of an electrical distribution system. An individual synchrophasor is typically a part of a larger network of units that communicate through Phasor Data Concentrators (PDCs) with a central control facility. The foremost goal of this case study is to demonstrate the validity of the actor network derived from the Simulink translation. Additionally, the validity of a Synchrophasor's behavior is critical to business and safety concerns of the public should they fail to detect a problem in the power grid.

The PMU modelled in Fig. 6.13 has five constant parameters and six outputs. These constants represent the chosen parameters for the sinusoidal signal representing the electrical

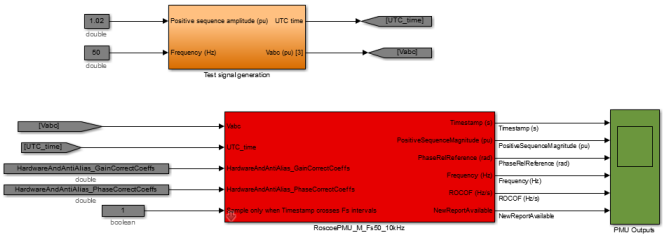

Figure 6.13: Roscoe Public PMU

signal and the filter coefficients for the measurement units. The top level system also includes two subsystems. The first is the PMU itself while the second is the *Test Signal Generation* that creates a sinusoidal signal and a timestamp for the PMU to sample.

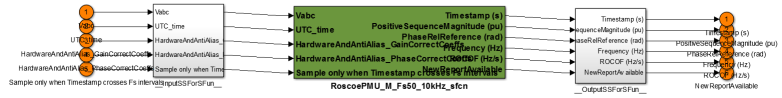

Figure 6.14: Simulink PMU

Within the PMU there are two additional subsystems and an SFunction. These subsystems pass their inputs directly through to their outputs, providing the signal typing information needed to statically verify the SFunction's interface, as discussed in section 5.2. The S-Function is a Simulink API wrapper that contains a precompiled *.MEX* file that was generated from an external C/C++ or Fortran file.

Figure 6.15: SFunction Type Filter

The result of this translation is shown in Fig. 6.16. While we represent it here graphically, the actual translation generates a textual MRICDF specification. The generated actor network is a single top level composite actor with no inputs and the six outputs of the PMU. In addition to the constant parameters and output port interface actors, this contains two composite actors representing the *test signal generation unit* and the composite actor representing the *phasor measurement unit*.

The five constant parameters become constant actors as the output blocks similarly become output actors. Meanwhile each system is translated into composite actors. The composition of the translated *Phasor Measurement Unit* is shown in Figure 6.19 while the internal composition of the test signal generation actor is shown in Figure 6.17.

The actors of the translation of the Test Signal Generator corresponds almost one for one with those in the original Simulink block diagram with the interface unchanged and operations

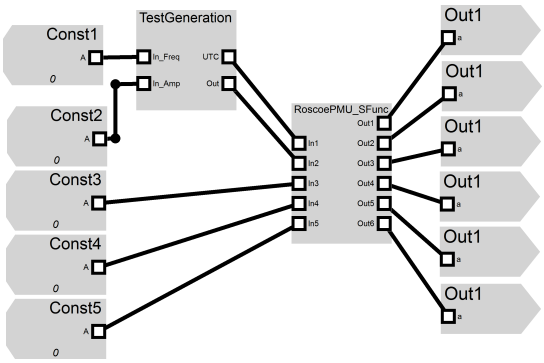

Figure 6.16: Roscoe PMU in MRICDF

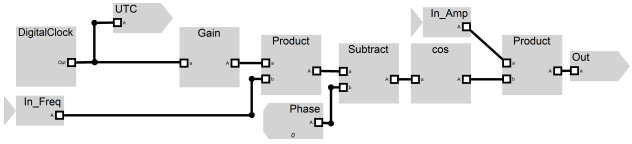

Figure 6.17: Test Generation in MRICDF

becoming function actors. The exception is the *Digital Clock* which has been translated as a composite actor that produces a numeric output at periodic intervals based on the parameters of the Simulink block.

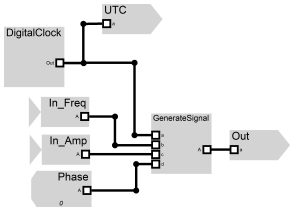

Figure 6.18: Manual Implementation of the Test Generator in MRICDF

By contrast, when manually implemented the size of the test signal generator is reduced each mathematical operation may be condensed into a single function actor. This is possible because there is no temporal state to be maintained between the operations being combined. If a *buffer* actor were present between the actors they would have to have remain separated to preserve the clock relations of the model.

Similarly the actor network generated from the Simulink PMU subsystem is a 1 for 1 translation from the original Simulink block diagram with the type filtering subsystems becoming composite actors and the S-Function being translated into an equivalent function that calls the same library function(s). This differs only slightly from the manual implementation in that when designing directly in MRICDF the signal filtering subsystems/composites are not required.

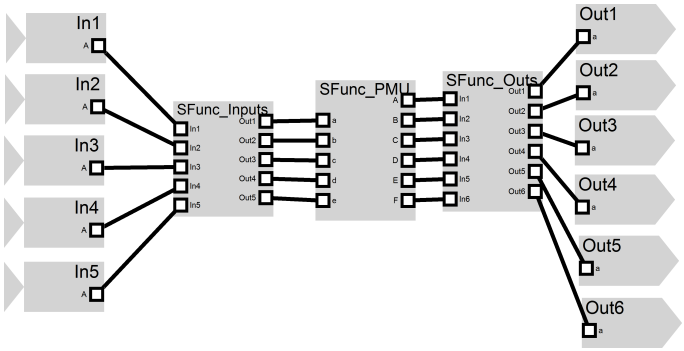

Figure 6.19: PMU Subsystem in MRICDF

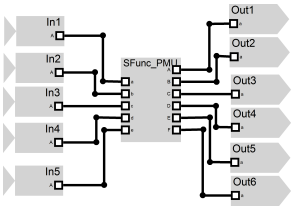

Figure 6.20: Manual Implementation of the PMU in MRICDF

Chapter 7

Conclusions and Future Work

Developing DREs is a time consuming and error prone task. If software multithreading is involved then there are additional development concerns regarding the determinism and reliability of the software's dynamic behavior. These issues are further complicated when the DRE in question is a safety critical system. In order to meet the pressures of industrial competition, developers must work swiftly to bring their products to market. However, to ensure their products meet safety standards, they must create and test numerous prototypes. Any prototypes that can't be refined and incorporated into the final system cost additional development time that risks being invalidated by a future design revision.

In this paper we present the beginnings of an end-to-end toolchain, *APECS* that will streamline the design flow process from prototype to final product. *AADL* forms the basis of our design framework. Its extensibility and capability to comprehensively describe a model, hardware and software, are crucial to the framework's ability to maintain a single model that is supported by all the necessary analyses techniques currently in the toolchain as well as any future extensions that may be made. Our direct contribution is the ability to formally specify the system's software with *MRICDF*, attaching these specifications to their corresponding components in the model through the *AADL*'s properties. These specifications are given at the process level, allowing us to leverage *MRICDF*'s multithreaded code generation capabilities. This approach frees the developers from a fine grained approach, automating the synchronization and validation of thread behavior for the system. *EmCodeSyn*, MRICDF's code synthesis framework, serves the *APECS* toolchain by providing this threading analysis. *Ocarina*, modified to accept AADL software with MRICDF specifications, generates software that runs on the target platform atop a high integrity middleware. APECS adds additional steps to *Ocarina*'s frontend that allow thread components to be dynamically added based on *EmCodeSyn*'s analysis. This constitutes the core of the *APECS* framework, but we have plans for a number of extensions.

Currently *Ocarina* can perform real-time schedulability analysis by invoking the AADL analysis tool *Cheddar* on the model. This requires that timing properties be manually

specified for each thread. In the future, we plan to include the work described in [44] in APECS. The inclusion of this work would allow for thread timings to be automatically optimized and extracted from *MRICDF* specifications for the dynamically generated threads.

Currently, *MRICDF*'s epoch analysis may only be applied to each process specification individually. While this allows for the formal verification and validation of each process in turn, we plan to eventually adapt the epoch analysis for APECS such that it may be extended to analyze the software of a system as a whole - including bus communications.

In order to better support existing models and to provide a more user friendly interface for commercial engineers, we further plan to add a Simulink [37] based alternative interface for APECS. Simulink is a popular tool for embedded software design. Unfortunately, its semantics are informal and vary depending on the simulator and user options selected. By extending APECS with the capability to automatically translate Simulink to MRICDF we will allow developers to use a language with which they're familiar and still obtain formally validated, distributed software.

Bibliography

[1] Compositional translation of simulink models into synchronous bip. Technical Report TR-2010-16, Verimag Research Report.

[2] Ravenscar. `http://www.adaic.com/standards/05rm/RM-Final.pdf`, 2005.

[3] AUTOSAR. `http://autosar.org/`, 2014.

[4] CORBA. `http://www.corba.org/`, 2014.

[5] Microsoft .NET Framework. `http://msdn.microsoft.com/en-us/library/zw4w595w.aspx`, 2014.

[6] Object Management Group. `http://www.omg.org/`, 2014.

[7] P. A. Abdulla and J. Deneux. Designing safe, reliable systems using scade. In *In Proc. ISoLA 2004*, 2004.

[8] Adele graphical editor. `https://wiki.sei.cmu.edu/aadl/index.php/Adele_Graphical_Editor`, 2014.

[9] AEEC. *Avionics Application Software Standard Interface Part 1-3, ARINC Specification 653P1-3*, 2010.

[10] A. Agrawal, G. Karsai, and F. Shi. A uml-based graph transformation approach for implementing domain-specific model transformations. In *International Journal on Software and Systems Modeling*, 2003.

[11] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electron. Notes Theor. Comput. Sci.*, 109:43–56, Dec. 2004.

[12] Arinc 653 plugin. `http://aadl.info/aadl/osate/osate-doc/osate-plugins/arinc653.html`, 2014.

[13] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[14] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sep 1991.

[15] G. Berry. Proof, language, and interaction. chapter The Foundations of Esterel, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.

[16] B. Berthomieu *, P.-O. Ribet, and F. Vernadat. The tool tina: Construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[17] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse France, 2008.

[18] Y. Bertot, P. Castran, G. i. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Donnes complmentaires http://coq.inria.fr.

[19] L. Besnard, E. Borde, P. Dissaux, T. Gautier, P. Le Guernic, and J.-P. Talpin. Logically timed specifications in the AADL : a synchronous model of computation and communication (recommendations to the SAE committee on AADL). Rapport Technique RT-0446, INRIA, Apr. 2014.

[20] P. Bieber, E. Noulard, C. Pagetti, T. Planche, and F. Vialard. Preliminary design of future reconfigurable ima platforms. *SIGBED Review*, 6(3):7, 2009.

[21] G. Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[22] O. Bouissou and A. Chapoutot. An operational semantics for simulink's simulation engine. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 129–138, New York, NY, USA, 2012. ACM.

[23] F. Boussinot and R. D. Simone. The esterel language. In *Proc. of the IEEE*, volume 79, no. 9, page 12931304, Sept. 1991.

[24] M. Bozga, V. Sfyrla, and J. Sifakis. Modeling synchronous systems in bip.

[25] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended aadl models. *Comput. J.*, 54(5):754–775, May 2011.

[26] M. Bozzano, A. Cimatti, M. Roveri, J. Katoen, V. Y. Nguyen, and T. Noll. Codesign of dependable systems: A component-based modeling language. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, pages 121–130, July 2009.

[27] Eclipse modelling framework. `http://www.eclipse.org/modeling/emf/`, 2014.

[28] M. Faugère, T. Bourbeau, R. De Simone, and S. Gerard. Marte: Also an uml profile for modeling aadl applications. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 359–364. IEEE, 2007.

[29] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st edition, 2012.

[30] K. Forsberg and H. Mooz. The relationship of system engineering to the project cycle. *INCOSE International Symposium*, 1(1):57–65, 1991.

[31] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[32] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In P. Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, 2011.

[33] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[34] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Softw. Eng.*, 18(9):785–793, Sept. 1992.

[35] T. Henzinger. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292, Jul 1996.

[36] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Trans. Embed. Comput. Syst.*, 7(4):42:1–42:25, Aug. 2008.

[37] T. M. Inc. Simulink. `http:/www.mathworks.com`.

[38] ISO/IEC. Annotated ada 2005 language reference manual. Technical Report 8652:2007(E) Ed. 3, 2006.

[39] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[40] B. Jose, J. Pribble, L. Stewart, and S. Shukla. Emcodesyn: A visual framework for multi-rate data flow specifications and code synthesis for embedded applications. In *Specification Design Languages, 2009. FDL 2009. Forum on*, pages 1–6, 2009.

[41] B. Jose and S. Shukla. An alternative polychronous model and synthesis methodology for model-driven embedded software. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 13–18, 2010.

[42] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science*, 9:1296–1321, 2003.

[43] F. Kordon and Luqi. An introduction to rapid system prototyping. *IEEE Transactions on Software Engineering*, 28(9):817–821, 2002.

[44] M. Kracht. Real-time emcodesyn. Master's thesis, Virginia Tech, Blacksburg, Virginia, Under Preparation.

[45] M. Le Borgne, H. Marchand, . Rutten, and M. Samaan. Formal verification of signal programs: Application to a power transformer station controller. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 271–285. Springer Berlin Heidelberg, 1996.

[46] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[47] Y. Ma, H. Yu, T. Gautier, P. Le Guernic, J.-P. Talpin, L. Besnard, and M. Heitz. Toward polychronous analysis and validation for timed software architectures in aadl. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1173–1178, 2013.

[48] Y. Ma, H. Yu, T. Gautier, J. Talpin, L. Besnard, and P. Le Guernic. System synthesis from aadl using polychrony. In *Electronic System Level Synthesis Conference (ESLsyn), 2011*, pages 1–6, 2011.

[49] S. Messaoud. Translating Discrete Time SIMULINK to SIGNAL. Master's thesis, Virginia Tech, Virginia, 2014.

[50] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla. A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications. In *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*, pages 21–30, 2013.

[51] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla. A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications. In *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*, pages 21–30, July 2013.

[52] V. Y. Nguyen, T. Noll, and M. Odenbrett. Slicing aadl specifications for model checking. In *NASA Formal Methods*, pages 217–221, 2010.

[53] Open source aadl tool environment - getting started. `http://www.aadl.info/aadl/currentsite/currentusers/open.html`, 2009.

[54] S. I. S. of Automotive Engineers. As5506/1 architecture analysis and design language annex (aadl), vol. 1, annex e: Error model annex. Technical report, SAE, 2006.

[55] U. of Pennsylvania MoBIES team. *HSIF Semantics*, 2002.

[56] M. Pantel and et al. The topcased project – a toolkit in open source for critical applications & systems design.

[57] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 200–209, Dec 1998.

[58] A. Roscoe, I. Abdulhadi, and G. Burt. P and m class phasor measurement unit algorithms using adaptive cascaded filters. *Power Delivery, IEEE Transactions on*, 28(3):1447–1459, July 2013.

[59] A.-E. Rugina, D. Thomas, X. Olive, and G. Veran. Gene-Auto: Automatic Software Code Generation for Real-Time Embedded Systems. In *DASIA 2008 Data Systems In Aerospace*, volume 665 of *ESA Special Publication*, page 28, Aug. 2008.

[60] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[61] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[62] SAE. Annex behavior specification v1.6. Technical Report AS5506, 2007.

[63] SAE. *Architecture Analysis and Design Language v2.0 (AS5506)*, 2008.

[64] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded system design for automotive applications. *Computer*, 40(10):42–51, Oct 2007.

[65] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar: a flexible real time scheduling framework. *INTERNATIONAL ACM SIGADA CONFERENCE, ATLANTA*, 2004.

[66] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time simulink to lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, Nov. 2005.

[67] Uml 2.x. `http://www.eclipse.org/modeling/mdt/?project=uml2`, 2014.

[68] Xtext. `http://www.eclipse.org/Xtext/`, 2014.

[69] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin. From {AADL} to timed abstract state machines: A verified model transformation. *Journal of Systems and Software*, 93(0):42 – 68, 2014.

[70] H. Yu, Y. Ma, T. Gautier, L. Besnard, P. L. Guernic, and J.-P. Talpin. Polychronous modeling, analysis, verification and simulation for timed software architectures. *Journal of Systems Architecture*, 59(10, Part D):1157 – 1170, 2013.

[71] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 221–228, Washington, DC, USA, 2008. IEEE Computer Society.