Model to code generation of UML/SysML activity diagrams for ARM CortexM MCUs

MohammadHossein AskariHemmat

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master Of Applied Science at

Concordia University

Montréal, Québec, Canada

April 2015

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **MohammadHossein AskariHemmat**

Entitled: **Model to code generation of UML/SysML activity diagrams for ARM CortexM MCUs**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science (Electrical & Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Zahangir Kabir Chair's name	Chair
Dr. Samar Abdi Examiner's name	Examiner
Dr. Mohammad Mannan Examiner's name	Examiner
Dr. Mounir Boukadoum Co-Supervisor's name	Co-Supervisor
Dr. Otmane Ait Mohamed Supervisor's name	Supervisor

Approved by ______________________________

Chair of the ECE Department

__________ 2015 ______________________________

Dean of Engineering

ABSTRACT

Model to code generation of UML/SysML activity diagrams for ARM CortexM MCUs

MohammadHossein AskariHemmat

The complexity in embedded systems has been increased in the last years. To overcome the system complexity various methodologies have been presented. Both in industry and academia, Model-Based design has been accepted to be the best solution to solve this problem.

Model-Based Design is a technique for developing embedded system applications and cyber-physical systems based on a hierarchy of reusable design blocks. SysML/UML activity diagrams are widely used for the modelling and analysis of complex systems, and they have become the de facto standard for software and embedded systems.

Previously in our group, we formalized SysML activity diagrams by developing a calculus called New Activity Calculus (NuAC). In this work, we redefined NuAC terms to support RTX (Keil Real-Time Operating System) and present an automated SysML/UML activity diagram to RTX code generator, using mapping rules expressed in NuAC.

To achieve this goal, we proposed a set of mapping rules that were used in mapping a SysML/UML activity diagram into a suitable code to be executed on ARM CortexM processor family. To automate the process of code generation, we presented a JAVA application that uses the proposed rules to automatically generate the RTX code from the input activity diagram model.

To demonstrate the capability of the developed tool, we use it to implement a train control algorithm on an ARM Cortex-M4 device. The implemented model is run on the target platform and the correct functionality of the system is verified.

To my loving family

ACKNOWLEDGEMENTS

First and foremost, I would like to thank the almighty ALLAH for his blessing to finish this research.
I would like to express my deepest gratitude to my supervisor, Dr. Otmane Ait Mohamed, for his excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research. This thesis would not have been possible without his guidance and support. I also want to thank Dr. Mounir Boukadoum, for co-supervising my research work and for giving his valuable comments on my research.
I would like to thank all my fellow researchers and colleagues in Hardware Verification Group (HVG) at Concordia University. Without their guidance, support and continual encouragements, this thesis would not have been possible.
I would like to take this opportunity to thank Regroupement Stratgique en Microsystmes du Qubec (ReSMiQ) for their financial supports. Without their financial supports I would not have been able to continue my research work.
Last but not least, I thank my family for their constant moral support and their prayers. They are the people who are closest to me and suffered most for my higher study in abroad. Their support was invaluable in completing this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

AATC	Advance Automatic Train Control
AF3	AutoFOCUS3
BART	Bay Area Rapid Transit
BSV	Bluespec SystemVerilog
CMSIS	Cortex Microcontroller Software Interface Standard
DCT	Discrete Cosine Transform
ESL	Electronic System Level
INCOSE	International Council on Systems Engineering
RTX	Real Time eXecutive
UML	Unified Modeling Language
MBSE	Model Based System Engineers
NuAC	New Activity Calculus
OMG	Object Management Group
OMT	Object Modeling Technique
PRISM	PRobabilistIc Symbolic Model checker
RTOS	Real-Time Operating System
SysML	Systems Modeling Language
WCSD	Worst Case Scenario Distance

Chapter 1

Introduction

1.1 Motivation

The complexity in embedded systems has been increased in the last years. New heterogeneous systems which combine different domains are more common. Aircrafts, automobiles, cell phones, medical equipment is an example where domains such as electronics, communication, software, mechanics, physics, mathematics and medicine are part of the systems development today. The International Council on Systems Engineering (INCOSE)[1] identified Model-Based Systems Engineering (MBSE) [2] as the key driver for effective and efficient system development in the future. Model-Based Design [3] is a technique for embedded system applications that reduces system complexity by creating a hierarchy of individual design blocks.

OMG(Object Management Group) Systems Modeling Language (SysML) [4] was developed in order to support effective communication among the parties involved by means of a standardized graphical notation. SysML [5] is a standard modeling language used for system applications. It reuses a subset of UML packages [6]. Mainly, it covers four aspects of system modeling: structure, behavior, requirement, and parametric diagrams. SysML is composed of several diagram types (use case, activity, sequence and so on). Particularly, SysML activity diagrams[9] are graphical representations of work-flows of step-wise activities and actions with support for choice, iteration and concurrency.

In order to execute the SysML model in an embedded hardware platform, the SysML model needs to be mapped to low level C code. In this thesis, our main goal is to generate an executable C code from a SysML activity diagram model. Particularly, we are interested to automatically generate code for ARM CortexM processor family [10] from a SysML activity diagram model. To achieve this goal, a set of mapping rules are proposed that maps a SysML activity diagram into a correct code to be executed on ARM CortexM processor family. In the following, we will present other alternative modeling languages/tools other than UML/SysML models. Finally, the proposed methodology and thesis contribution will be explained later in this chapter.

1.2 Related Works

1.2.1 MATLAB and Simulink based modeling

Most of the current available model to code generators, generate codes based on the Simulink models. For instance in [15], by using actor-oriented modeling language (System- MoC), the authors introduced a model transformation framework and claimed that they have closed the gap between classic ESL(Electronic System Level) [16] design flows and Simulink models. The proposed framework was integrated into an ESL design flow to reduce the development efforts. In [17] a framework for modeling, simulation and multi-platform code generation for sensor network algorithms based on Simulink models were introduced. They used StateFlow in Simulink to describe the protocol at a high level of abstraction. Later in their methodology, they used Matlab Embedded Coder to translate the high level models into the target C code. In addition to these methodologies, Matlab itself has Simulink Coder [18] and Embedded Coder [19] that can generate code based on the given Simulink model. However MATLAB and Simulink models are widely used in industry and academia, there are essential differences when they are compared to SysML and UML models.

First, MATLAB and Simulink do not have a standardized graphical notation whereas

SysML and UML have a standardized general purpose graphical notation for modeling different views of the system definition.

The second issue is that MATLAB and Simulink do not support inheritance-concepts for classification of components in order to enable their reuse whereas SysML and UML diagrams are highly compatible with object oriented programming concepts.

The last but not least major problem with MATLAB is that it is a commercial tool and the semantics of the models are proprietary to MATHWORKS. This make it very difficult to propose new modeling methodologies based on the MATLAB models.

1.2.2 Modeling based on AutoFocus3

Simulink based models are not the only alternatives to UML/SysML based modeling. Introducing new modeling language is another approach in Model-Based Systems Engineering. For instance, AutoFocus3(AF3) [20] is a model based development tool for distributed, embedded software systems. AutoFocus3 [20](AF3) is a model based development tool for distributed, embedded software systems. AF3 integrates all the required tools to design an embedded system from an input model. It is integrated with a NuSMV model checker [23] to perform the verification and the validation of the model before code generation. AF3 integrates modeling, testing and verification of an embedded system [38] in an Eclipse-based tool. It allows the user to define the model using three approaches namely State and Mode automata, Code Specification and Tabular Specification. AF3 provides model checking, boundary check analysis, reachability analysis and non-determinism checking of the system. It uses NuSMV as the model checking verification back end and supports most of the common temporal logic patterns. Verification artifacts and activities in AF3 are illustrated in Figure 1.1 [39].

AF3 is able to generate C-code based on the verified model. The correctness of the generated C code from an AF3 model has been proven in [40]. Three types of signals are supported in AF3: boolean; integer and double. AF3 has other features which can be found in [20]. However, AF3 suffers from two potential problems regarding the code generation

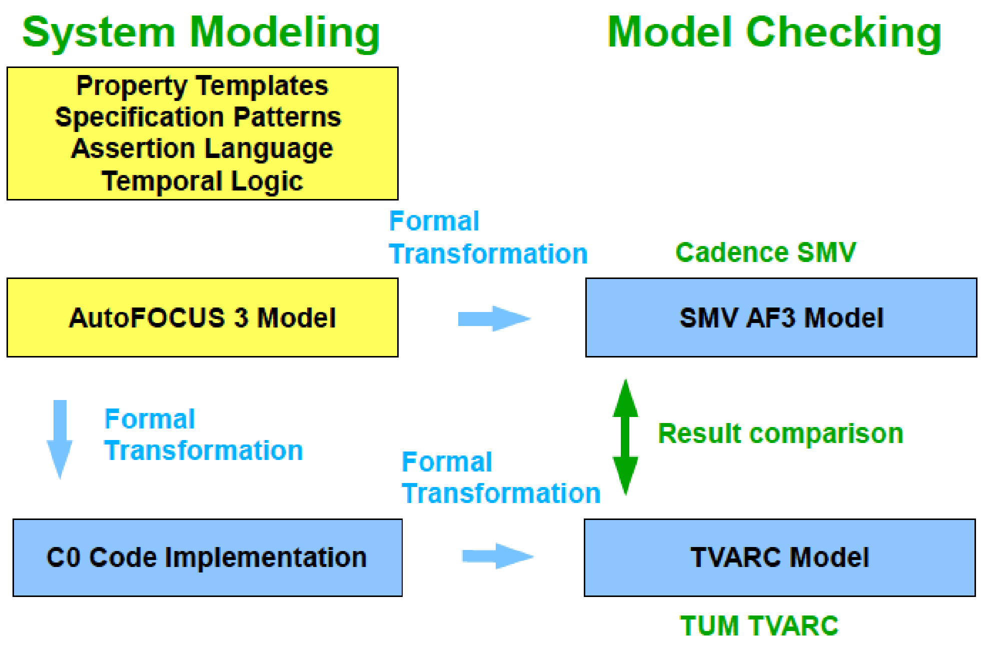


Figure 1.1: Verification artifacts and activities in AF3

process. The first problem is the lack of support for generating proper code to handle parallel processes. AF3 generates a separated C code for each individual block but there is no scheduler or an operating system to handle the parallel processes.

The second issue in modeling using AF3 is that the process of generating code, no specific platform is targeted. In order to run the code on an embedded hardware platform, the user needs to writer a wrapper around the generated code.

1.2.3 Modeling and Analysis of Real Time and Embedded systems

Modeling and Analysis of Real Time and Embedded systems also known as MARTE is the OMG standard for modeling real-time and embedded applications based on UML2 standard [42]. MARTE is a UML profile for real-time and embedded applications which provides support for specification, design, and verification. The goal of introducing this new profile is to replace the existing UML Profile for Schedulability, Performance and Time. MARTE consists in defining foundations for model-based description of real time and embedded systems [42].

The MARTE profile defines semantics for time and resource modeling. These semantics allow automatic transformations of models to lower abstraction level models such as UML for System On Chip (SoC) for hardware / software simulation or into C++ for implementation purposes [43]. MARTE does not offer any model verification solution, some analysis or the verification tools can be coupled with the modeling tool if the semantics of the models correspond to the semantics of the analysis or verification tool. Model transformation techniques can then be used to enable this coupling. Figure 1.2 illustrates the architecture of MARTE profile.

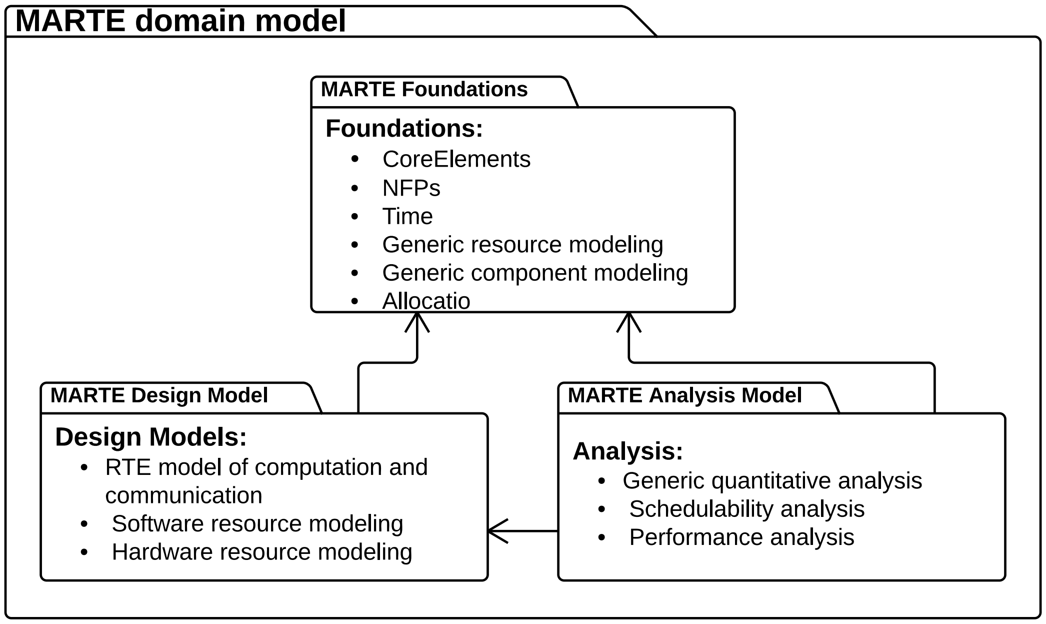

MARTE domain model
MARTE Foundations
Foundations:
• CoreElements
• NFPs
• Time
• Generic resource modeling
• Generic component modeling
• Allocatio
MARTE Design Model
Design Models:
• RTE model of computation and communication
• Software resource modeling
• Hardware resource modeling
MARTE Analysis Model
Analysis:
• Generic quantitative analysis
• Schedulability analysis
• Performance analysis

Figure 1.2: Architecture of MARTE profile [43]

The profile is structured around two concerns, one to model the features of real-time and embedded systems and the other one to annotate application models so as to support analysis of system properties. These are shown by the MARTE design model package in Figure 1.2, and the MARTE analysis model package, respectively. These two major parts share common concerns with describing time and the use of concurrent resources, which are contained in the shared package called MARTE foundations.

1.2.4 A Formal Verification Framework for BlueSpec System Verilog

In this work, a verification and implementation framework for mapping SysML activity diagrams to BlueSpec System Verilog(BSV) [21] is presented [25]. Bluespec SystemVerilog (BSV) is a declarative hardware modeling language based on a subset of SystemVerilog. It is used mainly in hardware designs specially in designing processors, memory subsystems, interconnects. It extends SystemVerilog by atomic rules and interfaces for state transitions, which can express concurrency easier. In this work, authors presented an efficient formal verification framework to improve the requirement checking of systems modeled by using SysML activity diagrams and synthesized as a BSV models. To verify these diagrams, they relied on PRISM [22] model checker. The mapping rules from SysML activity diagrams to PRSIM models were presented in [24]. They also proposed an efficient encoding algorithm to generate the correct BSV code proper to the verified SysML activity diagrams.

The most important limitation of this work that needs to be addressed is the lack of support for non-deterministic models. This problem shows itself more when the verified model needs to be implemented in a hardware platform. For instance, non-deterministic paths in SysML can be created by adding a probability on the outgoing edges of a decision node. Figure 1.3 illustrates such case described in SysML/UML activity diagram. In hardware these behaviors cannot be mapped to any resources. More ever, non-deterministic models are mostly used in verification and simulation phase.

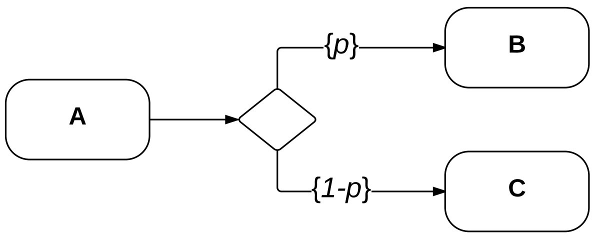


Figure 1.3: Probabilistic Decision in SysML activity diagram

1.3 Proposed Methodology

In this thesis, we are proposing a mapping tool which maps SysML/UML activity diagrams to an executable code for ARM CortexM based platforms. To achieve this goal, we have redefined the New Activity Calculus (NuAC) proposed in [24] and [25]. Specially, those rules which represent a non-determinism behavior (such as probability on a decision node) are redefined. Using the redefined NuAC terms, SysML/UML activity diagram artifacts are then formalized in a way that can express the Keil RTX [11] real-time operating system features. The ability to use our tool over different ARM CortexM platforms was one of the main challenges to overcome. The issue was addressed by using the ARM CMSIS Keil RTX real-time operating system. Since Keil RTX supports all ARM Cortex-M devices, the generated code can be executed on any platform that uses them.

Then, based on the SySML/UML activity diagram artifact formalization, a set of mapping rules are defined to map the SysML/UML activity diagram model to an executable code for ARM CortexM based platforms. The mapping rules were defined such that they could express the fundamental aspects of Keil RTX. In the process of mapping SysML/UML activity diagrams to RTX, we noticed that not all of the SysML/UML activity diagram artifacts have an equivalent function in RTX. For instance join two thread in SysML/UML activity diagram can be easily expressed by using the join node. Figure 1.4 illustrates expressing such behavior in SysML/UML activity diagram. However, in RTX such behavior is not defined as primitive function. To support such behavior, we had to define some new functions in RTX. In Chapter 3 we have provided the proper mapping rules for such cases.

To automate the process of generating RTX code from a SysML/UML activity diagram, a Java application has been developed. The mapping rules are used in the Java application to map each SysML/UML activity diagram artifact to it's corresponding RTX code. Chapter 3 also provides a pseudo code for this java application. Figure1.5 illustrates the overall mapping flowchart proposed in this thesis. This mapping tool is a part of a verification/Implementation framework which will be presented later in this thesis.

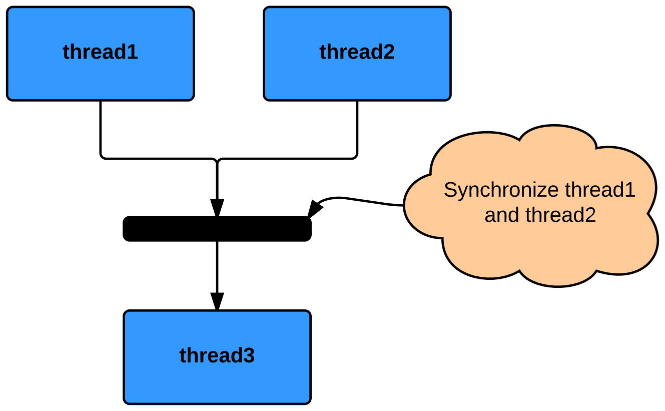


Figure 1.4: Join two thread in SysML/UML activity diagram

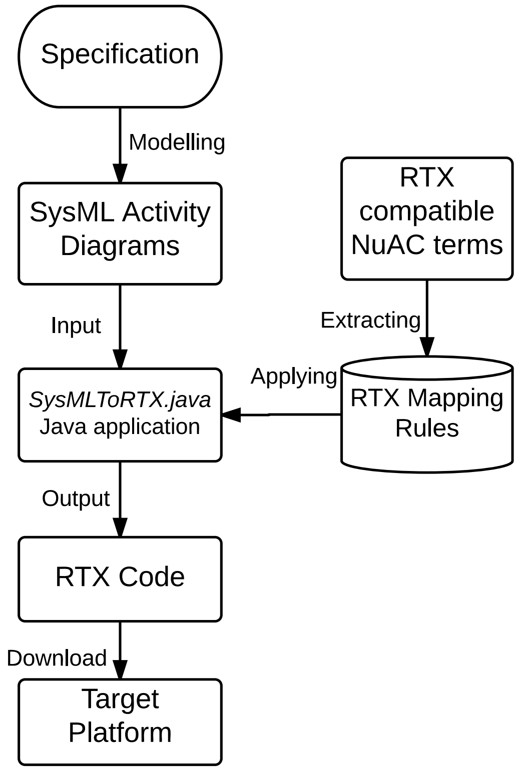


Figure 1.5: Mapping flowchart

1.4 Thesis Contribution

The contributions of this thesis can be summarized as follows:

- First, we extend the New Activity Calculus proposed in [24] and [25]. It was proposed for describing nondeterministic models in SysML, therefore no implementation code could be generated from those models. As discussed earlier, some of the original NuAC terms were redefined so that we would be able to generate the proper ARM CortexM executable codes.
- After redefining the New Activity Calculus, based on the redefined calculus, a set of mapping rules were defined. Mapping rules were defined in a way that can express the Keil RTX real-time operating system features. In the first stage, basic operating system features, like spawning threads and defining priority for threads, were added to the mapping rules. Later, more advanced features were added to the rules. For instance inter process communication was added to support sending and receiving events. Thread synchronization was also added to mapping rules by defining join function.
- Based on the mapping rules, a Java application was written to parse the SysML/UML activity diagrams. After parsing the model, the mapping rules were applied on the parsed model. The result is a C code ready to be compiled in Keil μVision[14] and then downloaded to any ARM CortexM platform.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

- In chapter 2, we present the preliminaries. The basic features of Keil RTX operating system and SysML/UML activity will be discussed by providing examples.
- In chapter 3, the mapping methodology will be presented. First the SysML/UML activity diagrams will be formalized and the New Activity Calculus will be redefined. Then, by using the redefined calculus, the mapping rules will be defined. Following in this chapter, the java application will be presented as a simple pseudo code.

- In chapter 4, two real world applications are presented and implemented following our mapping in an ARM CortexM4 platform. The first application is Bay Area Rapid Transit train controller system. The train controller algorithm will be presented in SysML activity diagram. Then, by utilizing our developed mapping tool, the executable code will be generated. After compiling the generated code in Keil μVision, the target platform will be uploaded with the train control system executable code. The result of running the code on the target platform will be presented later in the chapter.

  For the second experimental application, a JPEG encoder algorithm will be broken down into 5 different threads. Then a SysML model will be presented to handle the synchronization of these threads. Finally, by using the developed tool, the SysML model will be mapped to the Keil RTX codes.

- Chapter 5 provides the conclusion and future directions to our work.

Chapter 2

Preliminaries

This chapter introduces the fundamental background and the main concepts within the scope of this thesis. Section 2.1 provides an overview of SysML and UML language. This includes the main concepts and notations of UML/SysML design models . We particularly focus on activity diagrams syntax and semantics. Section 2.2 briefly presents the Keil RTX real time operating system. The features of RTX will be described briefly. Finally this chapter will be concluded in section 2.3.

2.1 SysML and UML modeling languages

2.1.1 Unified Modeling Language

UML stands for Unified Modeling Language and it had originally been built in order to serve modeling software systems. It is a result of the merging of three major notations: Grady Booch's methodology for describing a set of objects and their relationships [26], James Rumbaugh's Object-Modeling Technique (OMT) [27], and Ivar Jacobson's approach that includes ”use case” methodology[28]. It's maintenance and revisions are assumed by OMG[4] since 1997. It is a general-purpose visual modeling language that can be used for modeling standard software products, but also provides system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business processes and alike. Furthermore, the strength of UML resides in its wide acceptance by many industrial companies and the fact that it is non-proprietary, extensible, and supported by many tools and textbooks makes it a great modeling language for both academia and industry. UML is defined by using a meta-model, which is an abstraction of the UML model itself highlighting the properties of the language as well as the rules, constraints, and processes used to form the model. It offers a number of high-level modeling concepts allowing compact and abstract description of some systems properties. This abstractness capability offered by UML allows disregarding implementation details, which helps focusing on the essential business aspects of a solution. Furthermore, UML supports extension mechanisms (known as profiling mechanisms) such as constraints, stereotypes, and tagged values, which permit adapting it to a specific domain. A UML profile is a collection of extensions to the UML notations added for the purpose of tailoring the language to specific areas. Technically speaking, a UML model consists of elements such as packages, classes, and associations. The corresponding UML diagrams are graphical representations of parts of the UML model. UML diagrams contain graphical elements (nodes connected by paths) that represent elements in the UML model.

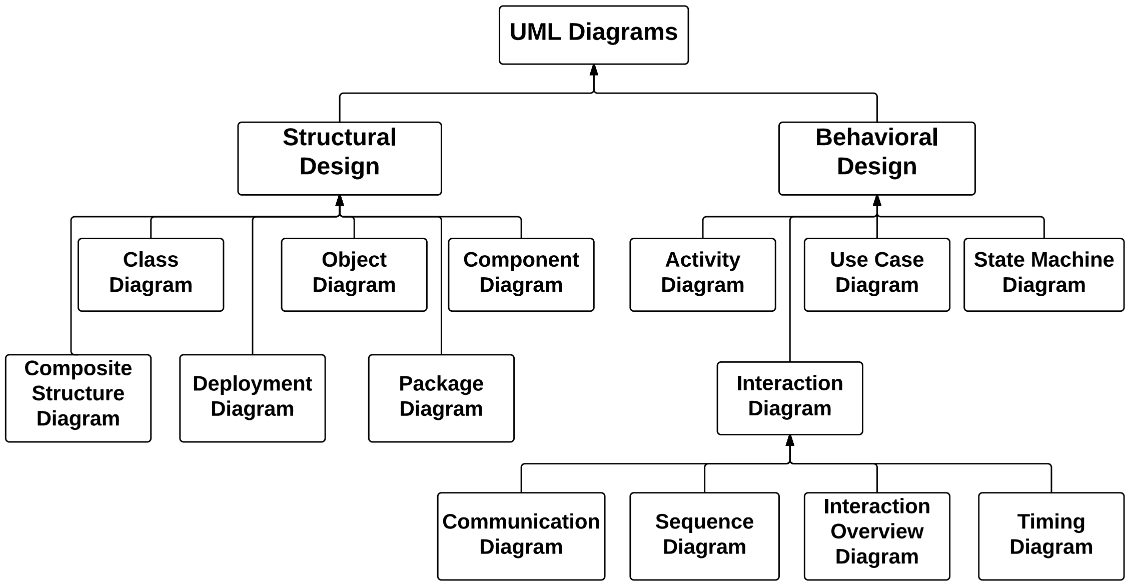


Figure 2.1: The taxonomy of UML structure and behavior diagrams [29]

In UML 2.0 there are 13 types of diagrams. The structural diagrams category includes class, component, composite structure, deployment, object, and package diagrams. These diagrams show the static features of a model such as classes, associations, objects, links, and collaborations. These features provide the skeleton in which the dynamic elements of the model are executed. On the other hand, behavioral diagrams contain activity, use case, and state machine diagrams as well as a sub-class named interaction diagrams including communication, interaction overview, sequence, and timing diagrams. They show the behavioral features and the functionality of a system as well as the interactions between objects and resources modeled in the structural diagrams. There exists a strong relationship between the diagrams themselves and between the behavioral and the structural models. This relationship constitutes the basis for consistency of UML models. The classification of UML 2.x diagrams is shown in Figure 2.1, reported from [4]. It highlights the diagram taxonomy differences with respect to the UML version. For example, new diagrams are proposed in UML 2.x such as composite structure, interaction overview, and timing diagrams. Others are updated compared to their UML1.x version like activity and sequence diagrams.

2.1.2 System Modeling Language

SysML [5] is a modeling language dedicated for systems engineering. It has it's roots in UML 2. Indeed, it is a UML profile that reuses some UML packages and extend others with system engineering specific features, in order to better fit the needed practices and methodologies. The mechanisms used in order to define these extensions are UML stereotypes, UML diagram extensions, and model libraries. The relationship between the two modeling languages UML and SysML is illustrated in Figure 2.2. The wide expressiveness of UML, its robustness and potential to be extended, as well as its large popularity have made it the best candidate to be customized for system engineering [31]. In the process of customization, various UML elements that are not required in systems engineering have been excluded such as components that are more dedicated to model software. Currently, SysML is gaining increased popularity and many companies from various fields such as defense, automotive, aerospace, medical devices, and telecom industries, are already using SysML, or are planning to switch to it very soon [31].

As SysML reuse subset of UML, it seems to be a good approach to describe its architecture with respect to UML as shown by Venn diagram in Figure 2.2, where UML and SysML are represented by two intersecting circles. Three areas of concern can be easily extracted from Figure 2.2, UML reused by SysML region, SysML extensions to UML region and UML not required by SysML.

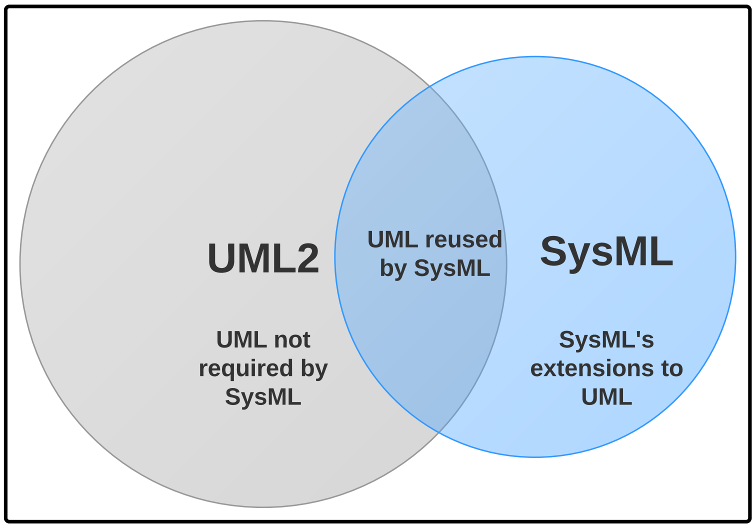


Figure 2.2: Relation between SysML and UML [32]

SysML is comprised of nine standard views/diagrams whereas UML consist of thirteen views/diagrams. Actually SysML retain some diagrams without modification while a number of diagrams are adopted with modifications. Furthermore SysML also introduce several new diagrams which are not present in UML. These diagrams are actually considered as extensions made by SysML. SysML diagrams are generally divided into three categories as shown by 2.3:

1. Diagrams that are used same as UML 2.0.

2. Diagrams used with slight modification from UML 2.0.

3. New types of diagrams.

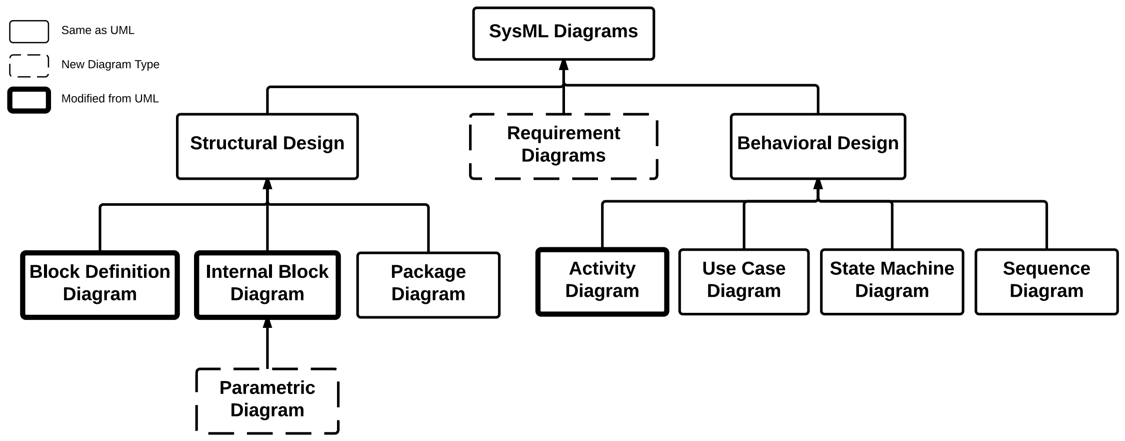


Figure 2.3: The taxonomy of SysML structure [30]

SysML extends UML by adding new diagrams such as requirement and parametric diagrams and integrating new specification capabilities such as embedding allocation relationships into design in order to represent various types of allocation, including allocation of functions to components, logical to physical components and software to hardware. Furthermore, it has fundamentally modified some other UML diagrams such as class diagrams since they were no more suitable to system engineering. Instead, block definition and internal block diagrams have been introduced in order to replace class and composite structure diagrams respectively.

2.1.3 Activity Diagrams

Initially, UML 1.x defines activity modeling using activity graphs that are endowed with a State chart-based semantics. Later, this concept has been modified with the release of UML 2.0, where activity graphs have been replaced with activity diagrams. More precisely, UML 2.0 activity diagrams are endowed with new semantics, which is independent of State charts semantics and supposedly based on Petri net semantics [4]. Generally, activity diagrams are used in modeling control flow and data-flow dependencies among the functions/processes that are defined within a system. They are widely used in computational and business processes modeling and use cases detailing. Basically, an activity diagram is composed of a set of actions related in a specific order of invocation (or execution) by control flow paths, optionally emphasizing the input and the output dependencies using data-flow paths. An action represents the fundamental unit of a behavior specification and cannot be further decomposed within the activity. An activity can be composed of a set of actions coordinated sequentially, concurrently or a combination of these. Furthermore, it may involve synchronization and/or branching. In order to enable these features, control nodes including fork, join, decision and merge can be used. They support various forms of control routing. Additionally, it is possible to specify hierarchy among activities using call behavior action nodes, which may reference another activity definition. The graphical artifacts corresponding to activity nodes and control flows are illustrated in Figure 2.4.

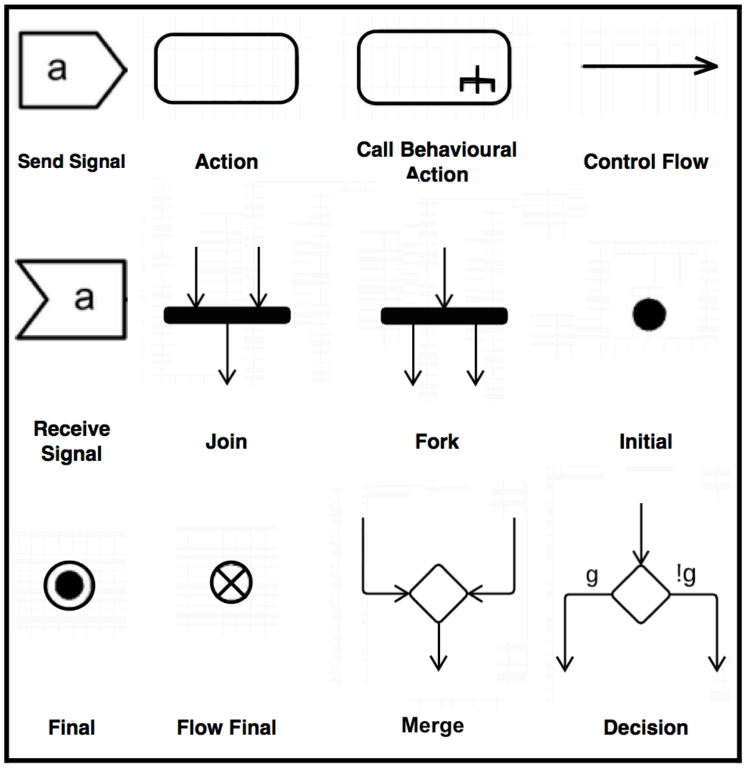


Figure 2.4: Activity Diagram Syntax

Concurrency and synchronization are modeled using forks and joins, whereas, branching is modeled using decision and merge nodes. While a decision node specifies a choice between two possible paths based on the evaluation of a guard condition (and/or a probability distribution), a fork node indicates the beginning of multiple parallel control threads. Moreover, a merge node specifies a point from where different incoming control paths have to follow the same path, whereas a join node allows multiple parallel control threads to synchronize and rejoin.

Activity diagrams behavior could be described in terms of tokens flow. The UML superstructure [4] specifies basic rules for the execution of the various nodes by explaining textually how a token can be passed from one node to another. At the beginning, a first token starts flowing from the initial node and moves from one node to the next one(s) with respect to the foregoing set of control routing rules defined by the control nodes until reaching either an final activity or a flow final node. As soon as a given action receives a token, it starts executing and when it terminates, the token is removed from the corresponding node and then offered to the node's output edges. In the case of a fork node, the incoming token is duplicated as many times as there are outgoing paths. With respect to the join node, the traversal of the token downstream on the outgoing edge requires that all needed tokens on the incoming edges are available and merged into one token. More specifically, the join node requires a particular ”join specification” requirement to be satisfied in order to issue a token on its single outgoing edge.

By default, this token traversal condition requires to have at least one token on each of the incoming edges of the join node. Finally, the first token that reaches an activity final node stops all the other active flows in the activity diagram. However, a token that reaches a final flow node ends only its corresponding control flow. As for SysML, apart from regular decision nodes, which describe choices between outgoing paths, it is possible to specify probabilistic behaviors in activity diagrams. There are two ways to use probabilities: On edges outgoing from a decision node and on output parameter sets (the set of outgoing edges holding data output from an action node). According to SysML specification, probability on a given edge expresses the likelihood that a token traverses this edge. An example of probabilistic decision node is shown in Figure 2.5.

For verification purposes, this feature(probabilistic behaviors) gives lots of benefits

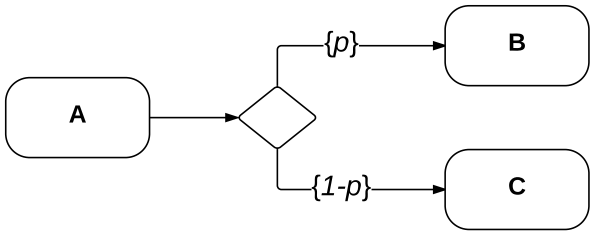


Figure 2.5: Probabilistic Decision in SysML activity diagram

to the verification team. However probabilistic behaviors are well supported in SysML, in this thesis we are not going to define any mapping rule to map probabilistic SysML models to RTX code. This decision makes sense since the implementation code cannot express any kind of probability. The generated code must act in a deterministic way to be considered as a correct implementation. In the next section, the Keil RTX real time operating system (RTOS) features will be presented.

2.1.4 SysML/UML based Modeling tools

To the knowledge of the author, there is not a complete tool which can provide all the steps required in Model Based System Design such as, modeling, verification, deployment and implementation. In [7], there is a list of the tools which allow creating system models based on SysML. The following is a list of the most important ones.

IBM Rational Rhapsody

Rhapsody is a commercial tool which lets system and software designers, to create real-time and embedded system model based on UML/SysML. This tool has the capability to do the trace between the stakeholder requirements and the elements of the model using its link with tools. Rhapsody also has an environment to simulate the model behavior which allows the designer to verify early their designs and also allows validating them with the stakeholder in an easy way.

Enterprise Architecture

Enterprise Architecture is also a commercial tool which was developed by Sparx Systems. It is focusing on the system modeling in UML and it has the capabilities to use SysML using a plugin developed by them. Unlike the previous tool, it only gives the elements to model in SysML, but it does not enable generating specific code to simulate the behavior of the system or to link the requirements from other tools to the requirements diagram in SysML. This tool is mostly used in the software development, although in the case of complex system it is too limited and it only can be used to document a system design.

Topcased

Topcased is an open source toolkit which is dedicated to system modeling for critical embedded systems [8]. This tool aims to help system designers and engineers to integrate formal-verification tools and generate code and documentation automatically. AdaCore, Airbus Air France, Anyware Technologies, Atos Origin, CNES, Laboratoire d'analyse et d'architecture des systmes and some other companies are the partners in the development of this project. This platform has model editors such as UML, SysML, AADL, SAM which are used to describe the specification of a system. By transformation, the system model can be verified by other simulation or analysis tools, which does the bridge between the model tools and the verification tools. The model can also be mapped to code, e.g. UML to C or UML to Java. Additionally, there is a tool which made the transformation from the model to natural language in order to create the supporting documentation to the development of the system.

In this work, we are going to use Topcased as our UML/SysML editor tool. The main reason behind our decision is that Topcased is released as an OpenSource tool. Later in this thesis, the Topcased Java libraries(available at [**?**]) will be used to parse the activity diagrams created in Topcased.

2.2 Keil RTX RTOS

The Keil RTX [11] is an easy to use Real Time eXecutive (RTX) providing a set of C functions and macros for building real-time applications. It is a professional-grade implementation of a real-time operating system written in C. It was originally designed for ARM7 TDMI and ARM9 Microcontrollers and later in version 4.2 the CMSIS-RTOS RTX was released to support ARM Cortex Microcontroller Software Interface Standard (CMSIS) [13]. The CMSIS-RTOS implementation is based on the Keil RTX Real-Time Operating System which is specifically designed for Cortex-M processor-based devices. In this work we are using the latest version of CMSIS-RTOS RTX namely version 4.75.

Keil RTX provides the capability of creating tasks dynamically (i.e. during execution time) and sets no restrictions on the number of tasks that can be created. It allows to use unlimited number of tasks each with 254 priority levels. Also unlimited number of mailboxes, semaphores, mutex, and timers are permitted. It provides a flexible scheduling which can be used based on the application. The following scheduling is supported by Keil RTX:

- Pre-emptive: Each task has a different priority and will run until a higher priority task is ready to run. This is commonly used in interactive systems where a device may be in standby or in background mode until some user data is provided.
- Round-Robin: Each task will run for a fixed period of CPU run-time (time slice). Data loggers/system monitors typically employ round-robin scheduling, all sensors or data-sources are sampled in turn with no prioritization.
- Co-operative: Each task will run until it is told to pass control to another task or reaches a blocking OS call. Co-operative multi-tasking can be seen in applications that require a fixed order of execution.

The benefit of using Keil RTX over other industrial RTOS such as FreeRTOS [33] is its deterministic behavior meaning events and interrupts are handled within a defined time. Also, the source code for Keil RTX is recently released. This can help developers to port Keil RTX on the new development platforms. The RTX kernel provides the following options for inter-process communication:

- Event flags: Event flags are the primary instrument for implementing task synchronization. Each task has 16 event flags associated to it. Hence, a task can selectively wait for 16 different events at the same time. In this case, the task can wait for all the selected flags (AND-connection), or wait for any one of the selected flags (OR-connection). Event flags can be set either by tasks or by ARM interrupt functions. Synchronize an external asynchronous event to an RTX kernel task by making the ARM interrupt function set a flag that the task is waiting for.

- Semaphores: If more than one task needs to access a common resource, special means are required in a real time multitasking system. Otherwise, accesses by different tasks might interfere and lead to inconsistent data, or a peripheral element might function incorrectly. Semaphores are the primary means of avoiding such access problems. Semaphores (binary semaphores) are software objects containing a virtual token. The kernel gives the token to the first task that requests it. No other task can obtain the token until it is released back into the semaphore. Since only the task that has the token can access the common resource, it prevents other tasks from accessing and interfering with the common resource. The RTX kernel puts a task to sleep if the requested token is not available in the semaphore. The kernel wakes-up the task and puts it in the ready list as soon as the token is returned to the semaphore. It is possible to use a time-out to ensure the task does not sleep indefinitely.

- Mutexes: Mutual exclusion locks (mutexes) are an alternative to avoid synchronization and memory access problems. Mutexes are software objects that a task can use to lock the common resource. Only the task that locks the mutex can access the common resource. The kernel blocks all other tasks that request the mutex until the task that locked the mutex unlocks it.

- Mailboxes: Tasks can pass messages between each other using mailboxes. This is usually the case when implementing various high level protocols like TCP-IP, UDP, and ISDN. The message is simply a pointer to the block of memory containing a protocol message or frame. It is the programmer's responsibility to dynamically allocate and free the memory block to prevent memory leaks. The RTX kernel puts the waiting task to sleep if the message is not available. The kernel wakes the task up as soon as another task sends a message to the mailbox.

2.2.1 RTX Threads

The CMSIS-RTOS RTX uses, by default, the Cortex-M SysTick timer to generate periodic interrupts for the RTX kernel timer tick. This periodic RTX kernel timer tick interrupt is used to derive the required time interval. CMSIS-RTOS RTX also provides configuration options for an alternative timer and tick-less operation [34].
To handle timeout and time delays for threads, the CMSIS-RTOS RTX thread management is controlled by the RTX kernel timer tick interrupt. The thread context contains all CPU registers (R0 - R12), the return address (LR), the program counter (PC), and the processor status register (xPSR). For the Cortex-M4 FPU and Cortex-M7 FPU the floating point status and registers (S0 - S32, FPSCR) are also part of the thread context. When a thread switch occurs, first, the thread context of the current running thread is stored on the local stack of this thread. Then the stack pointer is switched to the next running thread. Finally, the thread context of this next running thread is restored and this thread starts to run [34].

For Cortex-M4 FPU and Cortex-M7 FPU the thread context requires 200 bytes on the local stack. For devices with Cortex-M4 FPU and Cortex M7 FPU the default stack space should be increased to a minimum of 300 bytes [34].

The CMSIS-RTOS RTX employs a priority-based preemptive scheduler which ensures that from all the threads that are in the READY state, the thread with the highest priority gets executed and becomes the RUNNING thread. The CMSIS-RTOS assumes that threads are scheduled as shown in the Figure 2.6. Threads can share resources that are outside of the control of the RTX scheduler. This can prevent the highest priority thread

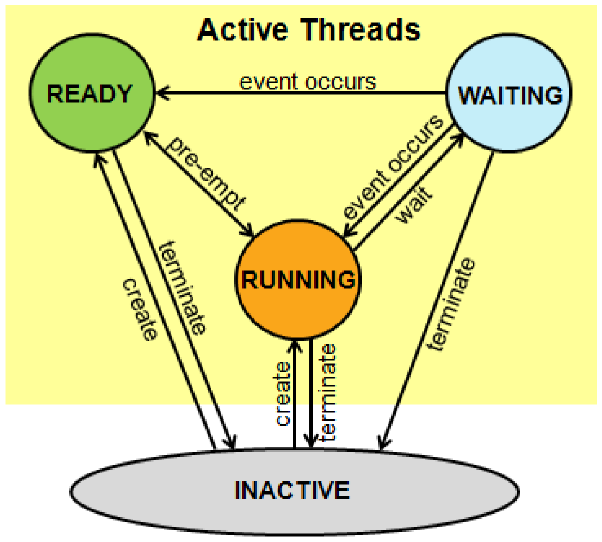


Figure 2.6: Thread State and State Transitions[34]

from running when it should. If this happens, a critical deadline could be missed, causing the system to fail. Priority inversion is the term of a scenario in which the highest-priority ready task fails to run when it should. Threads typically share resources to communicate and process data by using the CMSIS-RTOS Mutex Management. At any time, two or more threads share a resource, such as a memory buffer or a serial port, one of them may have a higher priority. As it will be presented later in this thesis, the developed mapping tool can only handle the send and receiving events. This is due to the fact that in SysML activity diagrams, there are no resources available to be assigned to mutex and mailbox. However, this issue can be solved by presenting a new SysML profile specifically designed for RTX which is out of the topic of this work.

Table 2.1 illustrates the technical information of CMSIS-RTOS RTX. For more information regarding CMSIS-RTOS RTX, please refer to [34].

2.3 Conclusion

In this section, a general information about system engineering was presented. Specially, a brief overview of two of the most popular system engineering languages were presented.

Table 2.1: Technical information of CMSIS-RTOS RTX[34]

Description	**Limitations**
Defined Tasks	Unlimited
Active Threads	250 max
Mailboxes	Unlimited
Semaphores	Unlimited
Mutexes	Unlimited
Signals	16 per thread
Timer Call back	Unlimited
Code Size	less than 5KB
RAM space for Kernel	300 B+ 128B Main stack
RAM space for Thread	StackSize + 52 Bytes
Hardware requirement	SysTick timer of other hardware timer
Thread Switch Time	less than 2.6 usec @ 72 MHz

From the different types of diagrams in SysML and UML, we picked activity diagrams. The reason behind choosing activity diagram from the other kinds is that activity diagrams are perfectly suitable in describing the flow of model. Also, concurrency and synchronization are well defined in activity diagrams. Later in this section, a brief overview of Keil RTX real time operating system was presented. The benefits of using Keil RTX over other real time operating system was provided. In the next section, the proposed mapping tool will be presented in details.

Chapter 3

Mapping Methodology

In this chapter the theory behind the proposed mapping tool will be discussed. We will begin by introducing our verification/implementation framework. The overall framework will be explained afterwards. Based on the proposed framework, the thesis contribution blocks will be explained. For that, we will start by formalizing the SysML/UML activity diagrams. This is done by utilizing and redefining New Activity Calculus (NuAC) presented in [24]. By using the redefined NuAC terms, the mapping rules will be presented. After defining the rules, a java application will be presented. This java application will utilize the defined mapping rules to map SysML activity diagrams to RTX code.

3.1 Implementation and Verification Framework

Figure 3.1 illustrates our verification and implementation framework. Our thesis contribution is also defined in this figure. In this chapter, each block of our thesis contribution will be explained. Figure 3.1 also show the previous works and our future work. As it was discussed earlier, our proposed mapping rule is based on a redefined version of the NuAC described in [24]. Redefining the original NuAC led into the incompatibility of verifying the SysML/UML activity diagrams using the tool presented in [24]. Proposing a compatible verification tool based on our redefined NuAC term is a future direction of this thesis. The overall architecture of such tool is illustrated in Figure 3.1. In the following sections, the thesis contribution blocks will be explained.

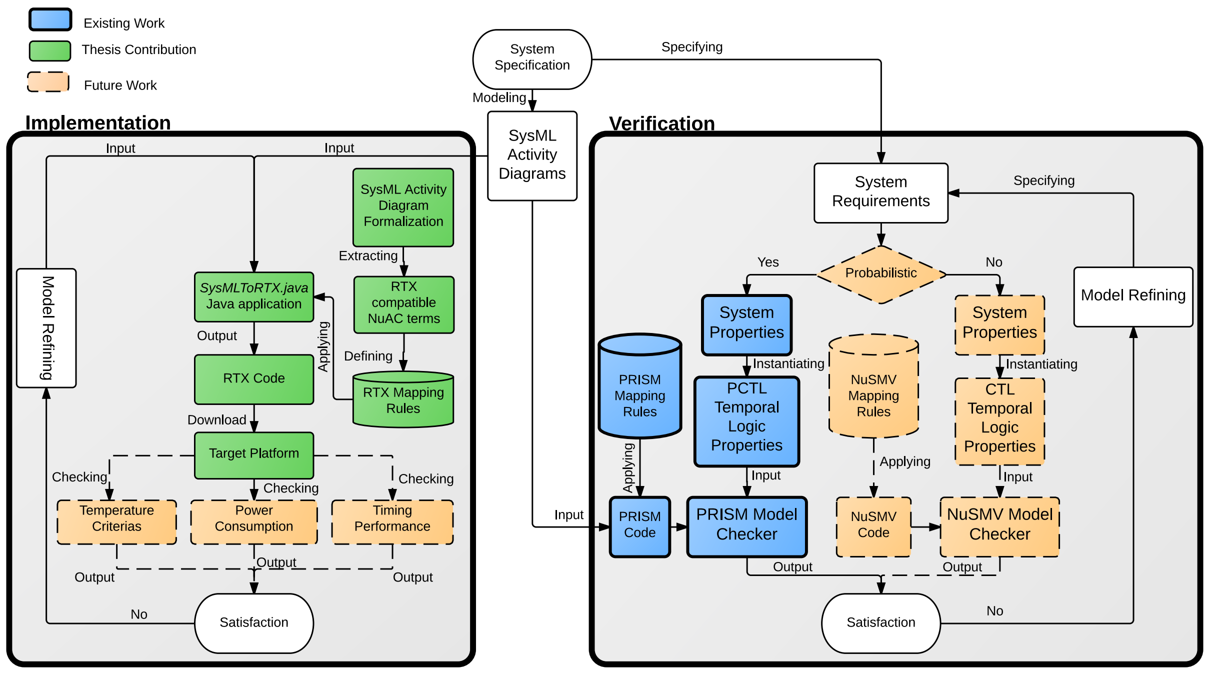


Figure 3.1: SysML to RTX verification and implementation framework

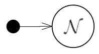
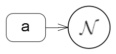
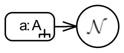
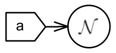
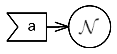
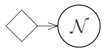
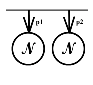
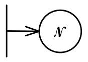
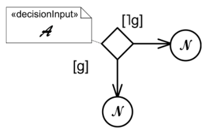
3.2 SysML Activity Diagrams Formalization

SysML/UML activity diagrams are graph-based diagrams in which activity nodes are connected by activity edges. A SysML/UML activity diagram includes three types of elements: activity nodes, activity control nodes and activity edges. In [25], the SysML/UML activity diagrams were formalized and rules were defined to map them to Bluespec System Verilog (BSV) descriptions. However, the notation used in [25] was for probabilistic models and no implementation resources could be assigned to the non-deterministic paths. In this work, we redefine mapping rules to include Keil RTX features.

In order to generate Keil RTX codes, the following changes were made in formalizing the SysML/UML activity diagram:

1- Only deterministic models can be mapped to Keil RTX

Table 3.1: Formalization of SysML Activity Diagram Artifacts.

Activity Artifacts	Formal Notation	Description
	$l : \iota \rightarrowtail \mathscr{N}$	Initial node is activated when a diagram is invoked
◉	$l : \odot$	Activity final node will terminate the execution of the diagram
⊗	$l : \otimes$	Flow final will kill the related token
	$l : a \rightarrowtail \mathscr{N}$	Action node defines an atomic action
	$l : a \uparrow \mathscr{A} \rightarrowtail \mathscr{N}$	Activity final node will terminate the execution of the diagram
	$l : a!v \rightarrowtail \mathscr{N}$	Send node is used to notify an event
	$l : a?v \rightarrowtail \mathscr{N}$	Receive node is used to wait for activation of an event (Blocking wait)
	$l : M(x_1, x_2) \rightarrowtail \mathscr{N}$	Merge node specifies the unconditional continuation of input flows , and x is the set of input flows x = $\{x_1, x_2\}$.
	$l : F(\mathscr{N}_1, \mathscr{N}_2, p_1, p_2)$	Fork node models the concurrency that begins multiple parallel control threads with priority of $p_1 and p_2$ respectively. UML 2.0 activity forks model unrestricted parallelism.
	$l : J(x_1, x_2) \rightarrowtail \mathscr{N}$	Join node allows the synchronization of threads, x is the set of input pins x = $\{x_1, x_2\}$.
	$l : D(\mathscr{A}, g, \mathscr{N}_1, \mathscr{N}_2)$	Decision node with a call behavior$\mathscr{A}$ and guarded edges {g, ¬g} .

2- Parallel processes generated after a fork should contain one of the thread priority defined in Keil RTX or by default the *Normal* priority will be assigned to the outgoing threads.

The NuAC syntax presented in Table 3.1 optimizes the syntax in [35] and [24] by eliminating the redundant terms. Also, NuAC exploits the commutativity and the associativity properties for multi-input/output nodes that are described by Property 3.1 and Property 3.2 in [24] respectively. These properties allow handling multiplicity by considering only two inputs/ outputs. Furthermore, NuAC covers more important behaviors such as: behavior calls, and communication by sending and receiving messages (signals or objects). Table 3.1 summarizes the new NuAC terms by showing the NuAC notation for SysML/UML activity diagrams. Based on these formal notation the following mapping rules were defined to map a SysML/UML activity diagrams to Keil RTX:

Listing 3.1: Generating Keil RTX Mapping Function

```
$\Gamma : \mathscr{A} \rightarrow \mathscr{R}$
$\Gamma(\mathscr{A}) = \forall n \in \mathscr{A}, L(n = \iota) = \top, L(n \neq \iota) = \bot,$
$Case(n)\ of$
    $l : \iota \rightarrowtail \mathscr{N} \Rightarrow$
        $in\{$
            $if(l = \top \& l_{\mathscr{N}} = \bot)$
                $l = \bot;$
                $l_{\mathscr{N}} = \top;$
            $else\ l_{\mathscr{N}} = l = \bot;$
        $\}$
            $\cup \Gamma(\mathscr{N})$
        $end$
    $l : M(x,y) \rightarrowtail \mathscr{N} \Rightarrow$
        $in\{$
            $if(l_x = \top \& l_{\mathscr{N}} = \bot)$
                $l_x = \bot$
```

```
                  $l_{\mathcal{N}} = \top;$
              $else\ if(l_y = \top \& l_{\mathcal{N}} = \bot)$
                  $l_y = \bot$
                  $l_{\mathcal{N}} = \top;$
              $else\ l_y = \bot \& l_x = \bot$
          $\}$
              $\cup \Gamma(\mathcal{N})$
          $end$
      $l : J(x,y) \rightarrowtail \mathcal{N} \Rightarrow$
          $in\{$
              $while(l_x = \top \& l_y = \top \& l_{\mathcal{N}} = \bot)\{$
              $\}$
                  $l_x = \bot$
                  $l_y = \bot$
                  $l_{\mathcal{N}} = \top;$
          $\}$
              $\cup \Gamma(\mathcal{N})$
          $end$
      $l : F(\mathcal{N}_1, \mathcal{N}_2, p_1, p_2) \Rightarrow$
      $in\{$
      $if(l = \top \& l_{\mathcal{N}_1} = \bot \& l_{\mathcal{N}_2} = \bot)$
          $l = \bot$
          $l_{\mathcal{N}_1} = \top;$
          $l_{\mathcal{N}_2} = \top;$
          $edge_1 = \{l\} \cap \{l_{\mathcal{N}_1}\};$
          $edge_2 = \{l\} \cap \{l_{\mathcal{N}_2}\};$
          $Crth(l, l_{\mathcal{N}_1}, \mathcal{N}_1, P_1);$
          $Crth(l, l_{\mathcal{N}_1}, \mathcal{N}_2, P_2);$
          $osThreadCreate(osThread(edge_1\_thread), NULL);$
```

```
        $osThreadCreate(osThread(edge_2\_thread), NULL);$
    $else\ l = \bot;$
    $\}$
    $end$
    $l : D(\mathscr{A}, g, \mathscr{N}_1, \mathscr{N}_2) \Rightarrow$
        $in\{$
            $if(l = \top \& \ g = \top \& l_{\mathscr{N}_1} = \bot)$
                $l = \bot$
                $l_{\mathscr{N}_1} = \top;$
            $elseif(l = \top \& \ g = \bot \& l_{\mathscr{N}_2} = \bot)$
                $l = \bot$
                $l_{\mathscr{N}_2} = \top;$
            $else\ l = \bot;$
        $\}$
            $\cup \Gamma(\mathscr{N}_1) \cup \Gamma(\mathscr{N}_2)$
        $end$
    $l : a \uparrow \mathscr{A} \rightarrowtail \mathscr{N} \Rightarrow$
        $in\{$
            $if(l = \top)$
                $l = \bot$
                $Cmth(l, l_{\mathscr{N}}, \mathscr{A}, \mathscr{N});$
                $L(\mathscr{A})\_method()$
                $l = \bot$
            $else\ l = \bot;$
            $\}$
            $\cup \Gamma(\mathscr{N})$
        $end$
    $l : \otimes \Rightarrow$
        $in\{$
```

```
                    $if(l = \top)$
                        $l = \bot$
                    $else\ l = \bot;$
                    $\}$
                    $\cup \Gamma(\mathscr{N})$
                $end$
        $l : a!v \rightarrowtail \mathscr{N} \Rightarrow$
                $in\{$
                    $osSignalSet(ID(v), v)$
                    $l = \bot$
                    $\}$
                    $\cup \Gamma(\mathscr{N})$
                $end$
        $l : a?v \rightarrowtail \mathscr{N} \Rightarrow$
                $in\{$
                    $osSignalWait(v, osWaitForever)$
                    $l = \bot$
                    $\}$
                    $\cup \Gamma(\mathscr{N})$
                $end$
```

Listing 3.2: Creating thread in context of Keil RTX

```
        $Crth(l, l_{\mathscr{N}}, \mathscr{N}, P) \Rightarrow$
        $in\{$
            $edge = \{l\} \cap \{l_{\mathscr{N}}\};$
                $void\ edge\_thread(void\ const\ *arg)\{$
                    $\Gamma(\mathscr{N})$
                $\}$
            $\chi = \chi \cup \{edge\_thread\_ID =$
```

```
                    $osThreadDef(edge\_thread, P, 1, 0)\}$
        $\}$
$\cup\Gamma(\mathscr{N})$
$end$
```

Listing 3.3: Creating Behavioral Function

```
$Cmth(l, l_{\mathscr{N}}, \mathscr{N}, \mathscr{A}_i) \Rightarrow$
$in\{$
    $void\ L(\mathscr{A}_i)\_method(IN(\mathscr{A}_i), OUT(\mathscr{A}_i))\{$
        $\Gamma(\mathscr{N})$
        $\}$
    $\lambda = \lambda \cup \{T(edge)\ L(\mathscr{A}_i)\_method(I(edge))\}$
    $\}$
$\cup\Gamma(\mathscr{N})$
$end$
```

The mapping function Γ presented in Listing 3.1 through Listing 3.3 generates the appropriate Keil RTX ($\mathscr{R}$) code from the input SysML/UML model ($\mathscr{A}$). In this mapping function, l represents the label of the current node. This label represents a boolean flag which will be activated based on the activation of the corresponding node. Initially, this flag is set to false except the initial node which is set to true. Generally, activating the node will result in activating the corresponding flag. For a node $n \in \mathscr{N}$ we defined function $L(n)$ which will return the label of it's related call behavior diagram.

RTX does not have a built-in *Join* function. In our mapping, we translate a SysML/UML join node to an infinite loop. The program will exit this loop only when both of its input threads are executed properly. We presented this mapping rule in line 25-34. l_x and l_y represent the label for input nodes. As mentioned earlier, initially, all the node's label are false except the initial node. Thus the hold will be taken from *while* $(l_x = \top \& l_y = \top \& l_{\mathscr{N}} = \perp)$ only if both of the input nodes are executed correctly.

Creation of parallel threads is accomplished by using a fork node. In RTX, defining a thread is done by calling $osThreadDef(name, priority, instances, stacksz)$. Creating a defined thread is done by calling $thread\_ID = osThreadCreate(osThread(name), NULL)$. The result of this operation is a handler to the thread which will be stored in $thread\_ID$ which can be used for the future references. The name of the thread is generated according to the fork's outgoing edges, which are obtained by applying the intersection on the fork's label set and its successors' labels. The result could be either $\{\emptyset\}$, meaning there is no edge which connects the fork to the corresponding node, or the corresponding edge which connects the fork to it's successor node. An edge is a set $\mathscr{E} \subseteq \{\mathscr{N}_1,\ \mathscr{N}_2,\ \mathscr{G},\ Prioirity\}$, where:

- $\mathscr{N}_1$ and $\mathscr{N}_2$ are the source and destination nodes
- $\mathscr{G}$ is the edge guard: $\mathscr{G} \subseteq \{true, false\}$
- Priority is defined when the source node is type of Fork node:
  $Priority \subseteq \{Idle,\ Low,\ BelowNormal,\ Normal, AboveNormal,\ High,\ Realtime\}$

In our mapping function, we have defined function $Crth(l, l_{\mathscr{N}}, \mathscr{N}, P)$ to create a thread. Based on the RTX syntax, this function will create a thread with the given priority P and edge name. All the created threads will be added to χ which is a list of all created threads. For call behavior nodes, the mapping rules has to be recursively applied on the call behavior activity. All call behavior nodes will result in creating a new method. This is accomplished by calling $Cmth(l, l_{\mathscr{N}}, \mathscr{N}, \mathscr{A}_i)$. The input and output variables of call behavior node are obtained by calling $IN(\mathscr{A}_i)$ and $OUT(\mathscr{A}_i)$ functions. The generated method will be added to the list of generated methods namely λ. Finally a call to generated function $(L(\mathscr{A})\_method())$ will execute the method in the context of RTX.
As mentioned earlier, events are also handled by our mapping tool. In RTX, to activate an event, the $osSignalSet(thread_id, event_name)$ function is used. Calling this function, will release the hold on any waiting event that is sensitive to the $event\_name$. In RTX, one can call $osSignalWait(event\_name, wait\_time)$ to wait for $event\_name$ to be activated. $ID(v)$ is

used to obtain the handler for a thread which contain the receive node namely v.

3.3 JAVA Application Unit

In this section, the developed JAVA Application tool will be discussed. To apply the rules on input model described in SysML/UML activity diagrams format, the model needs to be read and parsed. For that, we wrote a Java application to first read the model and then parse it in a way that can be fed to the mapping rules module.

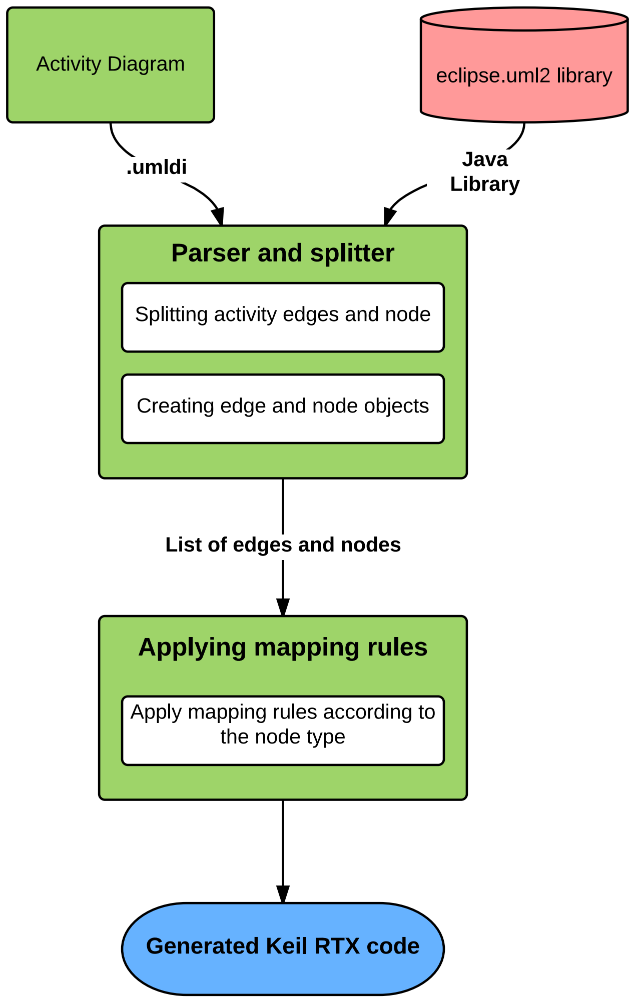


Figure 3.2: *SysMLToRTX.java* Java application

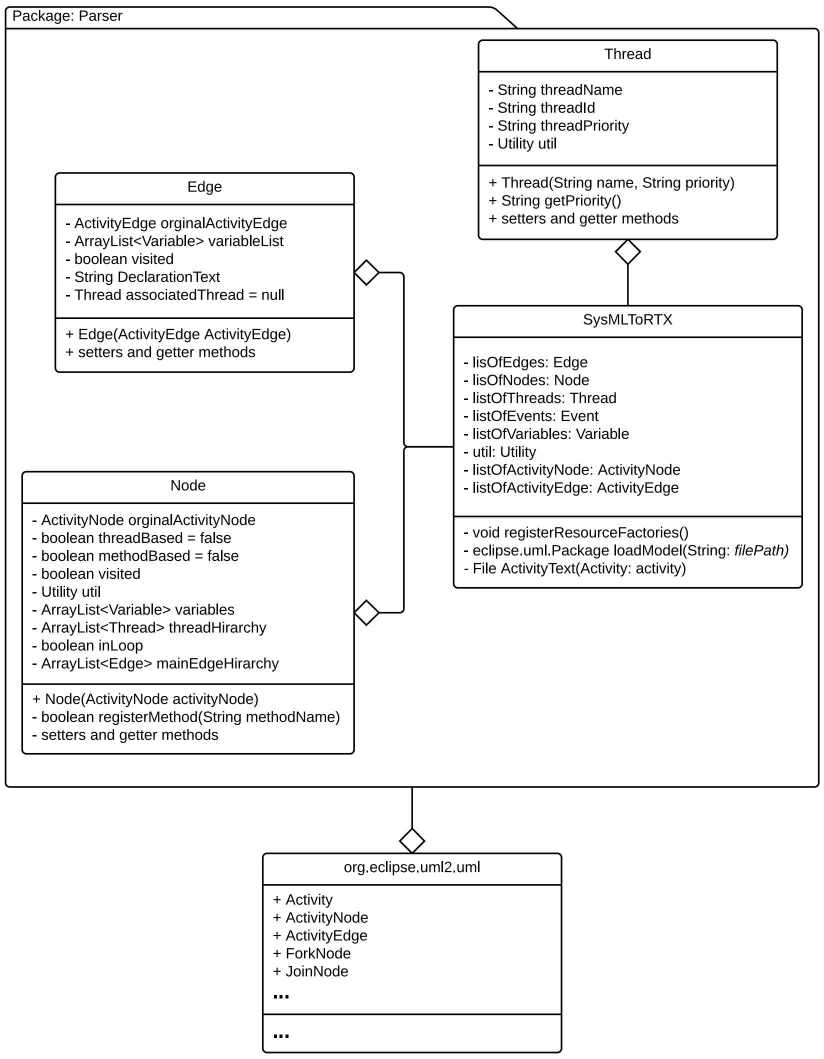

Package: Parser
Thread
- String threadName
- String threadId
- String threadPriority
- Utility util
+ Thread(String name, String priority)
+ String getPriority()
+ setters and getter methods
Edge
- ActivityEdge orginalActivityEdge
- ArrayList<Variable> variableList
- boolean visited
- String DeclarationText
- Thread associatedThread = null
+ Edge(ActivityEdge ActivityEdge)
+ setters and getter methods
SysMLToRTX
- lisOfEdges: Edge
- lisOfNodes: Node
- listOfThreads: Thread
- listOfEvents: Event
- listOfVariables: Variable
- util: Utility
- listOfActivityNode: ActivityNode
- listOfActivityEdge: ActivityEdge
- void registerResourceFactories()
- eclipse.uml.Package loadModel(String: filePath)
- File ActivityText(Activity: activity)
Node
- ActivityNode orginalActivityNode
- boolean threadBased = false
- boolean methodBased = false
- boolean visited
- Utility util
- ArrayList<Variable> variables
- ArrayList<Thread> threadHirarchy
- boolean inLoop
- ArrayList<Edge> mainEdgeHirarchy
+ Node(ActivityNode activityNode)
- boolean registerMethod(String methodName)
- setters and getter methods
org.eclipse.uml2.uml
+ Activity
+ ActivityNode
+ ActivityEdge
+ ForkNode
+ JoinNode
...
...

Figure 3.3: Simplified *SysMLToRTX* java application Class diagram

Figure 3.2 illustrates the *SysMLToRTX.java* Java application. The input model in activity diagram format will be fed to the parser and splitter module. The output of this module is a list of edges and nodes which were used in the input model. In the next module, the mapping rules will be applied accordingly.

A simplified *SysMLToRTX.java* Java application class diagram is illustrated in Figure 3.3. The parser package contains all the necessary classes. In Figure 3.3, only the important classes are illustrated. Other classes that are included in this package are: ***Event.java***, ***Utility.java*** and ***Variable.Java***. The ***Edge.Java*** and ***Node.Java*** classes are overriding the

eclipse.uml.ActivityNode and ***eclipse.uml.ActivityEdge*** classes for mapping the activity diagrams to RTX. The main class in this UML class diagram is SysMLToRTX class. The main function in this class is to first call two functions namely $registerResourceFactories()$ and $loadModel()$. These two methods are responsible to parse the input activity diagram and then create a $org.eclipse.uml2.uml.Activity$ object from it. This object is then passed to $ActivityText(Activityactivity)$ method. In $ActivityText(Activityactivity)$ method is where the mapping rules are applied. The output of this method is a file which contains the generated RTX code. A pseudo code of this method is presented in the following listings:

Listing 3.4: SysMLToRTX pseudo code

```
File ActivityText(Activity activity) :
String generatedCode
File outPutFile
for ActivityEdge aEdge in activity :
    Edge edge = new Edge(aEdge)
    Node node = new Node(aEdge.getSource())
    listofEdges.add(Edge(edge))
    listofNodes.add(edge.getSource())
    if (node instanceOf ForkNode) :
        node.SetThreadBased()
        listOfThreads.append(edge)
    elseif (node instanceOf DecisionNode) :
        if inLoop(node, activity) :
            node.setInLoop()
    elseif (node instanceOf JoinNode) :
        node.addIncomingThreads(node.getIncomings())
    elseif (node instanceOf CallBehaviorAction) :
        listOfMethods.add(node)
    elseif (node instanceOf SendSignalAction) :
```

```
20         listOfEvents.add(node)
21     elseif (node instanceOf InitialNode) :
22         node.setAsTop()
23     endif
24 endfor
25 for edge in listofEdges :
26 nodeObj = edge.getSource()
27     Case(nodeObj) :
28         instanceOf InitialNode :
29             if(!nodeObj.Visited())
30                 nodeObj.visitFlag() = true
31                 generatedCode.append(cmd)
32                 cmd = applyInitialNodeRules(edge.getSource())
33         ...
34     end
35 generatedCode.append(generatePrototypeForMethods())
36 generatedCode.append(generatePrototypeForThreads())
37 generatedCode.append(generateMethodDecleration())
38 generatedCode.append(generateThreadDecleration())
39 generatedCode.append(generateVarDecleration())
40 File.write(generatedCode)
```

The activity diagram (input model) will be first read by the java application. Then by using the *eclipse.uml2* java library(available at [**?**]) provided by *Topcased*, the activity diagram will be split into *eclipse.uml2.edge* and *eclipse.uml2.node* objects. This splitting is necessary in the next stage. The splitter part is presented in line 4-24. As mentioned earlier, we defined *Edge* and *Node* classes to be able to express the RTX features. The *Edge* constructor method accepts an *ActivityEdge* object. From that it will build an *Edge* object. The *Edge* object can express the flow of the program, spawn of a thread (if the source node is a Fork node) or simply a branch. As expressed in line 9, if the source node is a type of *ForkNode*, the outgoing edge would be considered as a thread. All threads are stored in *listOfThreads Arraylist* so that they can be referenced later in the program. If the source node is type of *DecisionNode*, the program has to check if the branch is causing a loop in the program. If that is the case, the source node *inLoop* flag will change the default value from false to true. If the source node is a type of *JoinNode*, the program will find the related threads of the incoming edges. These threads will be used later in the mapping rules. If the source node is a type of *CallBehaviorAction* node, it is required to create a method with same name of the node. The mapping rules will take care of creating the method, but the method prototype will be created in this stage. The method prototype will be generated by calling $generatePrototypeForMethods()$ function. As mentioned earlier, events are also handled by our mapping tool. If the source node is a type of *SendSignalAction* node, the node will be added to the *listOfEvents* list. Finally if the source node is a type of *InitialNode*, the node *TopActivity* flag will be set to true so that the main C function can be built inside this Activity.

In the process of applying rules, the model edges are read one by one from top to bottom. The source and target of an edge can be easily obtained by calling $edge.getSource()$ and $edge.getTarget()$ functions. To keep track of applying rules, the node *visitFlag* will be set to true. Finally, the rules for corresponding node will be applied and the generated code will be appended to the rest of the program.

After the mapping is done, the developed tool will generate a Keil μvision project. The project ,which contains the mapped code, can be later compiled and the generated machine code can be easily downloaded to the target platform.

3.4 Summary

In this chapter, a technical review of the developed tool was presented. The theory behind our mapping tool was explained. The mapping rules were presented in NuAC terms. This was achieved by, first formalizing the SysML activity diagrams. We presented a pseudo code of the *SysMLToRTX.java* Java application. The *SysMLToRTX.java* application is responsible to, first read the model and then parse the given activity diagram. After that the parsed activity is passed to the splitter were the activity is broken into an array list of edges and nodes. Finally by using the output of splitter and the mapping rules, the activity diagram is mapped to RTX code.

Chapter 4

Application

4.1 BART Case Study

This section describes the BART case study, that will be used as a model to be implemented on an ARM CortexM4 device using our developed tool. The following section pick-up pieces from the Bay Area Rapid Transit (BART) system to illustrate the possibilities of using the presented methodology on a real life application.

4.1.1 BART system overview

This section contains an informal description of a portion of the Advance Automatic Train Control (AATC) system being developed for the Bay Area Rapid Transit (BART) system. BART provides commuter rail service for part of California's San Francisco bay area. Specifically, this section contains those aspects of BART that are necessary to control the speed and acceleration for the trains in the system. Other aspects in BART control such as communication error recovery, routing, right-of-way signaling are out of the scope of this section. The overall objective of this case study is to construct a system within the infrastructure given, that can control the speed and acceleration of trains in the system subject to the various constraints that are described in the specification.

BART provides a heavy commuter rail service; on a typical work day, it serves around

250,000 passengers. During commute hours, over 50 trains can be in service. The system is controlled automatically and the on-board drivers have a limited role to play during normal operation, which includes signaling the system when the platforms are clear so that a train can depart a station. With a few minor exceptions, the BART system consists of double track: one track going one direction and one track going the other. The trains go from a starting point to an ending point (i.e., the track is not a loop). The trains that are moving on the tracks have to obey the speed and acceleration grade limit. Figure 4.1 illustrates the map of the BART transit system. The BART transit system is consist of 5 different track. In this casestudy, we are considering only Dublin/Pleasa track. This track is illustrated in Figure 4.2.

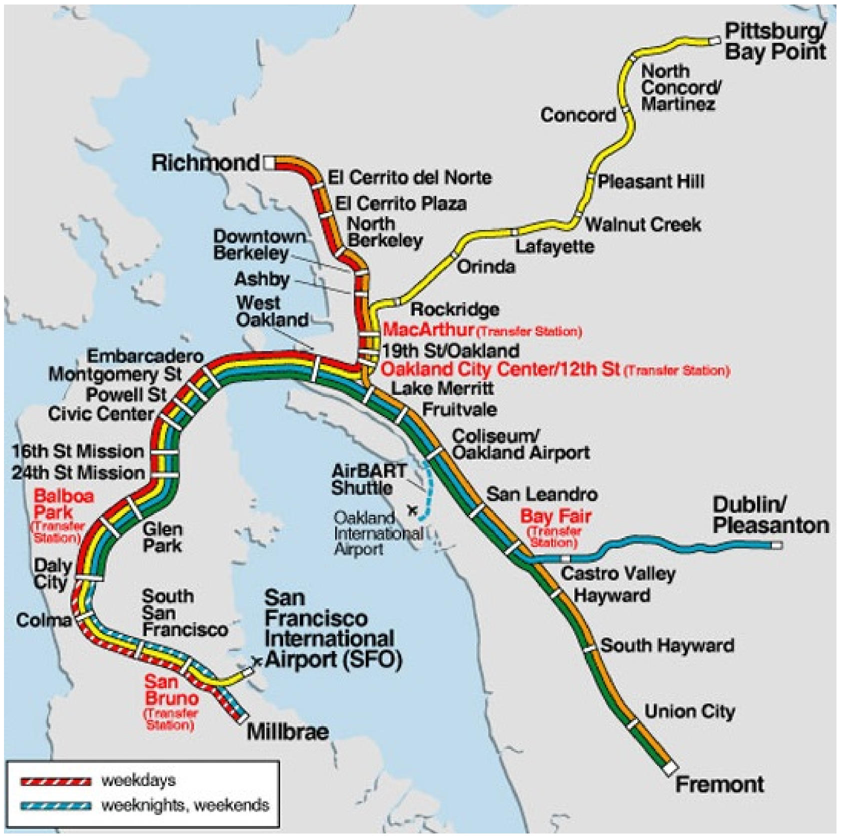


Figure 4.1: The map of the BART transit system

In our case study we chose the *Dublin* to *Daly City* track. This track is consist of 5 segments. Each segment contains speed and acceleration grade limit. Also, this track contains 18 gates. Each gate can be in *open* or *close* state. Table 4.1 illustrates *Dublin* to *Daly City* properties. The distances are calculated from the beginning of the track so for instance *FTVL(33273.5m)* means that FTVL gate is located 33273.5 meters from the beginning of the track. The data provided in Table 4.1 are obtained from BART's Geospatial data base [36] in Keyhole Markup Language (KML) format. Figure 4.2 illustrates the *Dublin* to *Daly City* track in Google map. The segments are represented in different colors and the gates are represented by 📍.

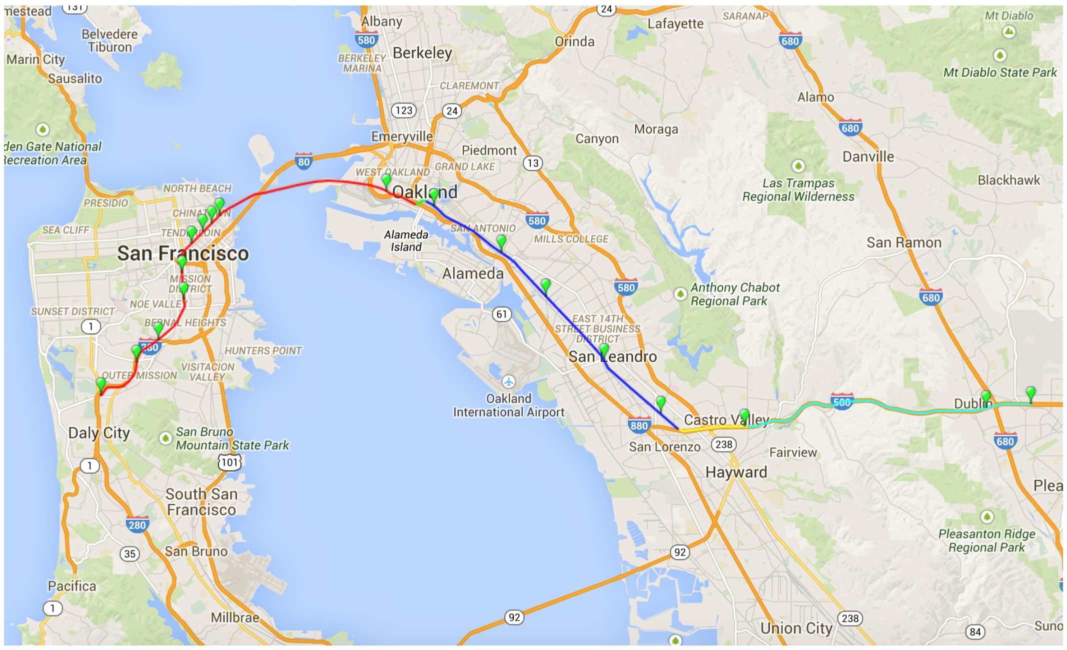


Figure 4.2: The Dublin/Pleasa track in BART transit system.

As mentioned earlier, the main objective is to control the speed and acceleration of the train. Speed and acceleration of a train has to be selected so that:

- The train should never get so close to a train in front that if the train in front stopped suddenly the following train would hit it.
- The train should not enter a closed gate
- The train should stay below the maximum speed of the track(defined by *CivilSpeed*)

Table 4.1: *Dublin* to *Daly City* track properties

Track Name	Segments	Segment Range	Gates	Civil Speed	Grade
Dublin to *Daly City*	DUBL_CAST_E	0km - 16.0km	DUBL (27.6m),WDUB(2447.7m)	36	0.8
	CAST_E_BAYF_S	16.0km - 19.7km	CAST(16129.5m)	80	0.3
	OAKY_BAYF_S	19.7km - 38.1km	BAYF(21396.1m), SANL(25428.4m), COLS(30028.6m), FTVL(33273.5m), LAKE(37714.1m)	70	3.49
	OAKY_SE	38.1km - 38.7km	No gates in this segment	50	1.00
	OAKY_DALY	38.7km - 62.6km	WOAK(40577.9m), EMBR(50023.7m), MONT(50577.6m),POWL(51441.5m), CIVC(52306.3m), 16th(53932.1m),24th(55350.1m), GLEN(58038.2m), BALB(59855.1m), DALY(62690.6m)	36	0.8

In our case study a Station Computer, which is a part of the Advance Automatic Train Control (AATC) system, controls the trains in their immediate area by giving speed and acceleration commands to the trains. Station computer runs the control algorithm for each train and will calculate the speed and acceleration accordingly. We assume the station computer has a direct access to speed, acceleration and position of the trains. The output of the algorithm is commanded speed (between 0 and 80Kmph) and acceleration (-2 to -0.45 Kmphps in braking state and 0 to 3 in propulsion) for a given train.

To simplify the model, we have abstracted the communication link between computer and trains and we assume that the commands from Station Computer are correctly received by the trains. Also we assume that the interlocking system does not close a gate when it is too late for an approaching train to stop. Trains are abstracted as a single location in the track. The operation of a train is modeled as follow:

Listing 4.1: Modeling the operation of a train

```
$let\ delta = 0.5$
$let\ grade = (-21.9 * currentSegmentGrade)/100$
$then:$
    $let\ n = nosePosition + v \times delta + \frac{1}{2}a \times delta^2$
        $+ \frac{1}{2} \times grade \times delta^2$
    $if(v == 0\ and\ vcm == 0):$
        $nosePosition = nosePosition$
    $else:$
        $nosePosition = n$
    $let\ speed = v + \frac{1}{2}a \times delta + \frac{1}{2}grade \times delta$
    $if(v == 0\ and\ vcm == 0)\ or\ speed <= 0:$
        $v = 0$
    $else:$
        $v = speed$
    $let\ noseAtNext = nosePosition + v \times delta$
```

```
16          $+\frac{1}{2} \times a \times delta^2$
17          $+\frac{1}{2} \times grade \times delta^2$
18          $if(v == 0\ and\ vcm == 0):$
19              $a = 0$
20          $elseif((v > (vcm - 2)\ and\ acm > 0)\ or$
21              $(v > (vcm - 2)\ and\ acm < 0)):$
22              $a = (21.9 \times grade)/100$
23          $else:$
24              $a = acm$
```

Where *acm* and *vcm* are the received commanded acceleration and velocity respectively and *delta* is the time in seconds between each command. The variable *grade* holds the acceleration due to grade (line 2). In line 4-9, by means of appropriate physical formulas, the next position of the train is calculated. If v and *vcm* are both 0 the position remains unchanged. In line 10-14 the train's velocity is calculated. Since the train cannot go backwards, the velocity will be set to zero if v is negative.

The acceleration is calculated in line 15-23. If the speed has achieved the commanded speed within the range of $\pm 2Kmph$ the trains attempts to maintain the current speed by compensating the acceleration due to grade.

While trains are trying to achieve their commanded velocity and acceleration goals, they have no notion of speed and acceleration limit of the segment. Also, stopping at or passing by a gate is not determined locally by the train. Thus, it is always the station computer that guides the trains and prevents potential catastrophes. Listing 4.2 illustrates a simple control algorithm for the Station Computer:

Listing 4.2: Station Computer Algorithm

```
1 $let\ t \in trainList$
2 $let\ delta = 0.5$
3 $let\ grade = (-21.9 * currentSegmentGrade)/100$
4 $let\ range = (WCSD(t) \times 2 + 230)$
```

```
then :
    $nextStopDistance = calcNextStop(t, trainList, gateList)$
    $segment, vcmCivilSpeed = CivilSpeed(t, range)$
    $if((nextStopDistance - t.position) < range):$
        $vcm = 0$
    $else:$
        $vcm = vcmCivilSpeed$
    $d1 = nextStopDistance - t.position$
    $if d1 < 0:$
        $acc = train.a() + 0.5$
    $else:$
        $acc = \frac{vcmCivilSpeed^2 - t.v^2}{2 \times d1} - grade$
    $if(acc < 0 \text{ and } acc > -0.45):$
        $acmCivilSpeed = -0.45$
    $else:$
        $acmCivilSpeed = acc$
    $d2 = nextStopDistance - t.position - WCSD(t)$
    $acc = \frac{-t.v^2}{2 \times d2} - grade$
    $if(nextStopDistance - t.position) > range:$
        $acmNextStop = t.a + 0.5$
    $else:$
    $if(acc < 0 \text{ and } acc > -0.45) and$
    $(d2 > ((t.v \times delta) + 0.5 \times grade * delta^2):$
        $acmNextStop = 0$
    $elseif(acc < 0 and acc > -0.45) and$
    $(d2 <= ((t.v \times delta) + 0.5 \times grade * delta^2):$
        $acmNextStop = -0.45$
    $else:$
        $acmNextStop = acc$
```

```
34        $if acmCivilSpeed < acmNextStop:$
35            $acm = acmCivilSpeed$
36        $else:$
37            $acm = acmNextStop$
```

The above algorithm will generate the appropriate velocity and acceleration for the given train. In line 4 the *range* and Worst Case Scenario Distance (WCSD) is calculated. The worst Case Scenario profile has been thoroughly explained in [37]. Table 4.2 and 4.3 illustrates the Worst case train parameters and Worst case train calculations receptively.

nextStopDistance is calculated based on the current position of the train and it's front train (if any) and if a closed gate is within the stopping range. A decision will be made based on which of the two is closest. These calculation will be made by passing the *train*, *trainList* and *gateList* objects to the *calcNextStop* function. The *civilSpeed* function will calculate the next segments in range and it will return the lowest *civilSpeed* of all. Also it will return the corresponding segment object. In line 9-12, the commanded velocity will be calculated.

The calculation of *acm* (commanded acceleration) is done in line 12-37. First the *acmCivilSpeed* is calculated. The calculation of *acmCivilSpeed* is only restricted by civil speed. The *acmCivilSpeed* is calculated so that the train reach a speed 2mph below *vcmCivilSpeed*. If the train happens to be already on that segment, *acmCivilSpeed* is set to the current acceleration incremented by 0.5Kmph. If the resulting acceleration is between 0 and -0.45(which is not allowed), it will be rounded off to -0.45. Next, *acmNextStop* is computed. If the next stop is out of range it will be set to the current acceleration incremented by 0.5 Kmph. Otherwise it will be set so that the train stop WCSD feet before the obstacle. Since the WCSD is shrinking while the train is getting closer to the obstacle, the train will stop at a reasonable distance to the obstacle. If necessary, the resulting acceleration will rounded off to be within the allowed acceleration range mentioned before. Finally the commanded acceleration is set to the minimum of *acmCivilSpeed* and *acmNextStop*.

Parameters	Description
NOSE	Estimated train nose location
PUF	Uncertainty Factor
PU	Position Uncertainty reported as one standard deviation
V_{CM}	Commanded Speed
AD	AATC Delay = 2 seconds
J_P	Jerk Limit in Propulsion
A_P	Acceleration in Propulsion
T_{JP}	Jerk Time in Propulsion = 1.5 seconds
A	Acceleration due to Grade = $-21.9 mphps * \frac{grade\ in\ \%}{100}$
MC	Mode Change
NCAR	Number of Cars = 10 Cars
NFAIL	Number of failed cars = 2 Cars
NFSMC	Number of cars in FSMC = 2 Cars
J_B	Jerk Limit in Braking = -1.5 mphps ps
BRK	Design Brake Rate = -1.5 mphps for exposed track and -2.0 mphps for covered track
FSMC	Fail Safe Mode Change Time = 8.5 Seconds

Table 4.2: Worst Case Scenario Distance parameters

4.1.2 Simulation and Verification in AF3

Based on the above algorithms, we modeled the system in AF3 to verify the correct functionality of the modeled system. Figure 4.3 illustrates the BART system in AF3. In this

Variable	Definition
D1	$NOSE + (PUF \times PU)$
D2	$V_{CM} \times AD$
J_P	A_P / T_{JP}
D3	$V_{CM} * T_{JP} + \frac{1}{2}Ap \times (T_{JP})^2 + \frac{1}{6} \times J_P \times (T_{JP})^3 + \frac{1}{2}A \times (T_{JP})^2$
V3	$V_{CM} + Ap \times T_{JP} + \frac{1}{2}J_P \times (T_{JP})^2 + A \times T_{JP}$
D4	$V3 \times MC + \frac{1}{2}A \times MC^2$
V4	$V3 + A \times MC$
Q_{FSMC}	$(NCAR - NFAIL - NFSMC)/NCAR$
T_{JB}	BRK/J_B
D5	$V4 \times T_{JB} + \frac{1}{6} \times JB \times Q_{FSMC} \times T_{JB} + \frac{1}{2}A \times T_{JB}$
V5	$V4 + \frac{1}{2}JB \times Q_{FSMC} \times (T_{JB})^2 + A \times T_{JB}$
T6	$FSMC - T_{JP} - MC - T_{JB}$
BFS	$BRK * Q_{FSMC}$
D6	$V5 \times T6 + \frac{1}{2}BFS \times (T6)^2 + \frac{1}{2}A \times (T6)^2$
V6	$V5 + BFS \times T6 + A \times T6$
Q	$(NCAR - NFAIL)/NCAR$
D7	$((VF)^2 - (V6)^2)/2 \times (BRK \times Q + A)$
WCSD	$\sum_{i=1}^{7} D_i$

Table 4.3: Worst Case Scenario Distance calculations

example we considered two trains are moving in the same direction in *Doublin* track. The Monitor block, illustrates the velocity, acceleration and the position of the trains. Figure 4.4 illustrates the *TrainController* module modeled in AF3. Each of the modules inside the *TrainContoller* modules are defined using the Code Specification [41] feature in AF3. The

AF3 project is available at [44].

For simulation and verification, the initial train positions play a key role in avoiding obstacles. In Simulation, the initial positions are calculated in a way that does not violate the safety regulations. For instance, we chose them to be much greater than the WCSD to any heading obstacle. As for verification, the properties are verified only if the initial position is more than the WCSD to the heading obstacle.

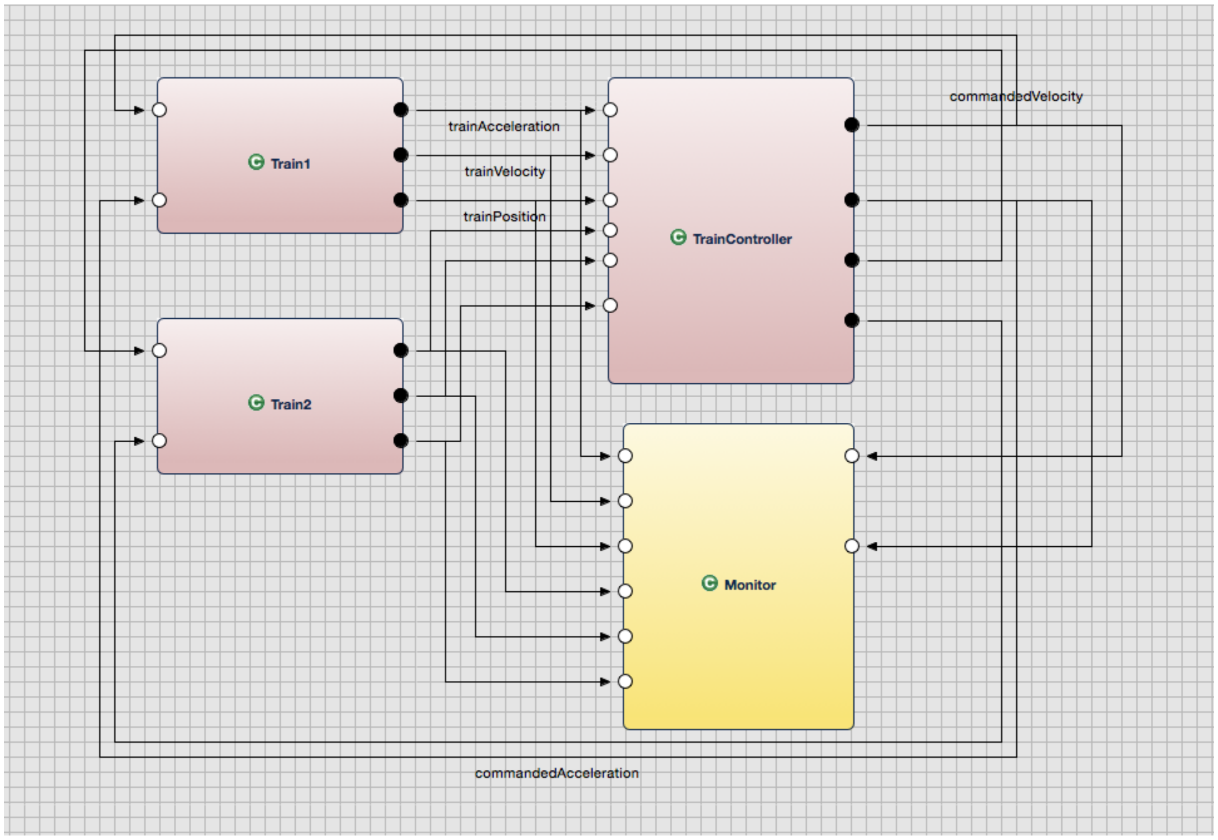


Figure 4.3: BART system modeled in AF3

AF3 uses NuSMV as an external tool to verify the properties of the model. The following are some of the properties that were verified:

$\forall\, i \in [1, 2, 3..., 18]$

P$_1$: $AG((T1_{initpos} - T2_{initpos}) > WCSD \& (T1_{initpos} - gate.begin_i) > WCSD)) \rightarrow AF(T1_v < T1.Segment_{CivilSpeed})$

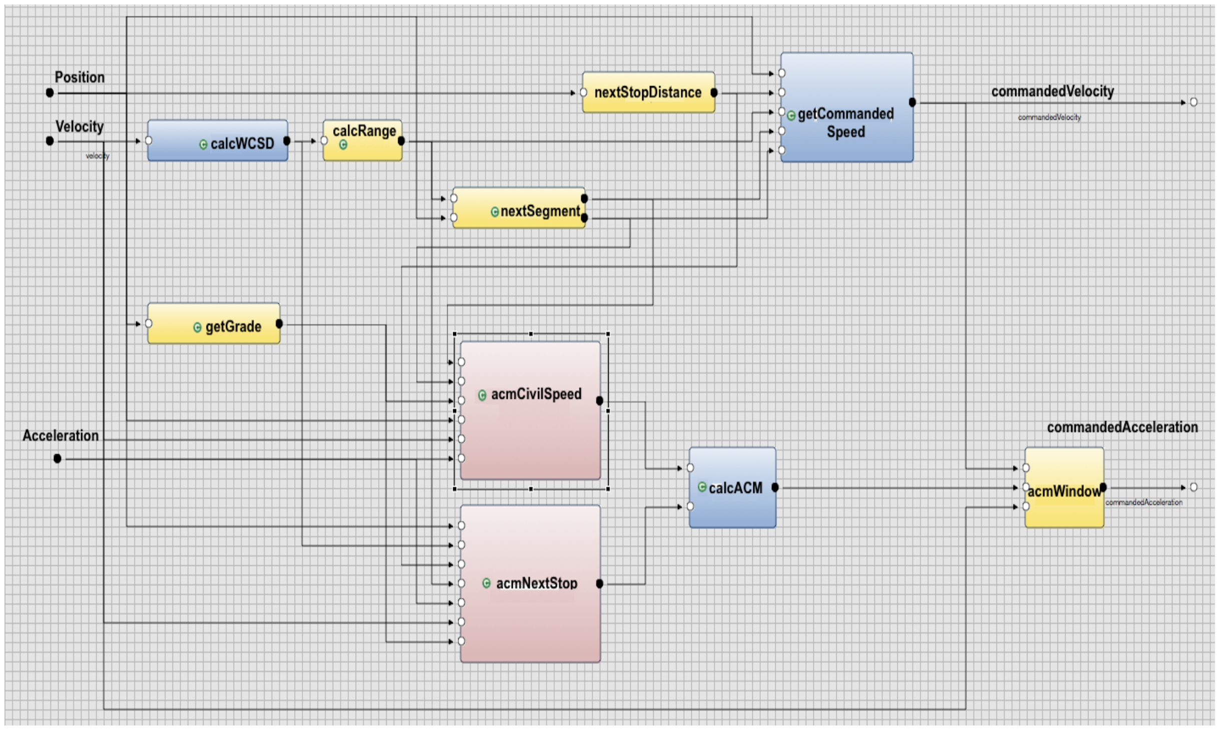


Figure 4.4: TrainController module in AF3

P$_2$**:** $AG((T1_{initpos} - T2_{initpos}) > WCSD \& (T1_{initpos} - gate.begin_i) > WCSD)) \rightarrow AF((T1_{pos} - T2_{pos}) < 0)$

P$_3$**:** $AG((T1_{initpos} - T2_{initpos}) > WCSD \& (T1_{initpos} - gate.begin_i) > WCSD) \& (T1_a < 0) \& (gate.state == 0)) \rightarrow AF((T1_{pos} - gate.begin_i) < 0)$

The first property (**P**$_1$) shows that if the initial position($T_{initpos}$) is selected appropriately(as discussed earlier), eventually, the train speed will never exceed the segment *CivilSpeed*.

The second property (**P**$_2$) shows that by selecting a proper initial position, the distance between two train will be never less than the WCSD.

The third property(**P**$_3$) shows that if a train is in braking state ($a < 0$) and it is approaching a closed gate ($gate.state == 0$), by selecting a proper initial position the distance between a train and any gate will be never less than 0(avoiding collision with a gate).

4.1.3 Implementation

In this section, with the mean of the proposed tool, the train control algorithm of AATC's station computer will be implemented in an ARM CortexM4 platform. The main goal is to implement the train control system in a way that satisfy the safety properties which were defined earlier.

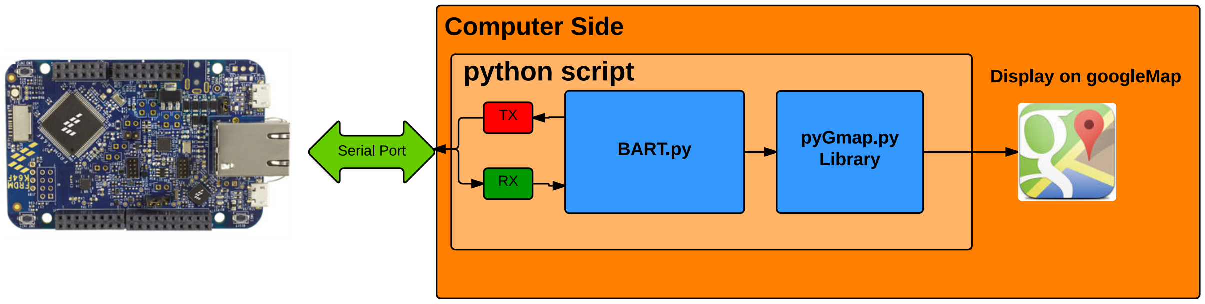


Figure 4.5: Overview of verification platform

To achieve this goal, we built a verification platform that verifies the correctness of the output commands. For that, we made a golden system that contain all the required parts in AATC including station computer,trains, segments and tracks. An overview of verification platform is illustrated in Figure 4.5. The BART system including station computer,trains, segments and tracks are modeled as a python script. The *BART.py* is responsible to communicate with FRDM-K64F Freescale freedom platform [45] over serial port. The commands from the board are sent back to PC as a packet. The packet format is illustrated in Figure 4.6 where $< CR >$ is the carriage return character.

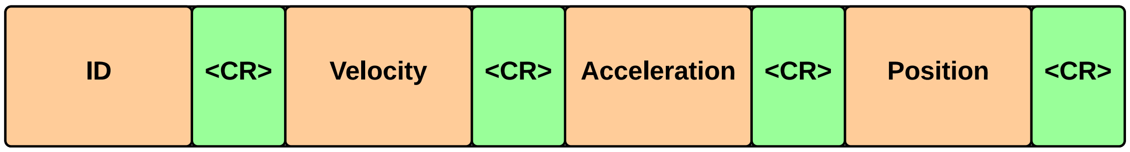


Figure 4.6: Communication Packet Format

As a response to this packet, *BART.py* will sent back the current position of the trains and will update the trains data(acm and vcm) accordingly. To implement the $ComputerStation$

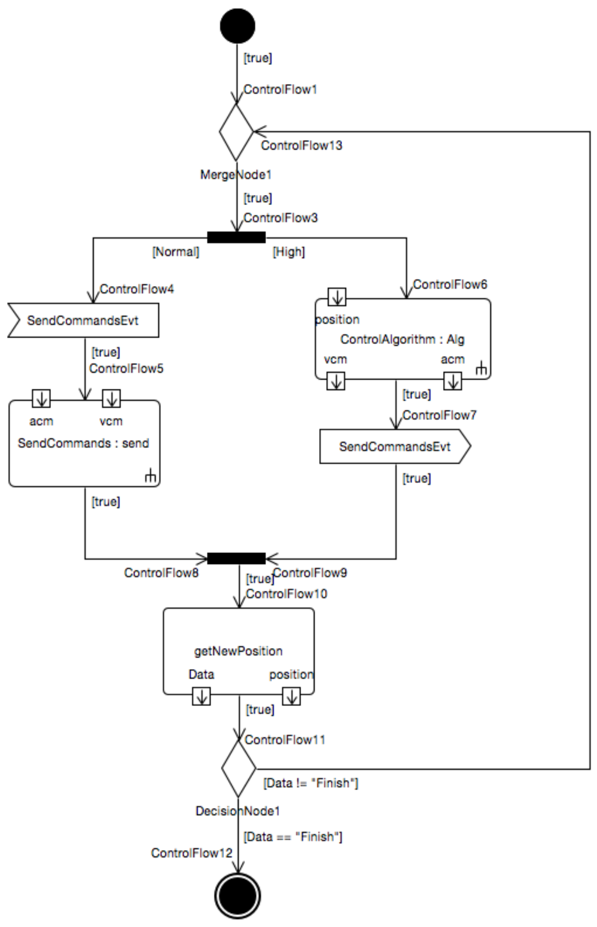


Figure 4.7: Abstracted Computer Station model

algorithm in the FRDM board, we modeled the system in Topcased using activity diagrams. Figure 4.7 illustrates an abstracted model of the implemented system. Listing A.1 illustrates the generated RTX code from the input model. Listing A.1 shows that after a successful system initialization, the RTOS will spawn two threads, one with a *Normal* priority and one with a *High* priority. The reason for different priority is that we want to give the priority *ComputerStationAlgorithm* Behavior action to be executed so it can then notify the send *waitForNewCommand* event and which will then lead to the execution of sending new command to PC. After each round, the system will wait for a response from PC to either quit or continue running the algorithm(by sending the train position). If the received command is *Finish* it means that there is no need to execute the algorithm any more so the *FinalNode* method will be called.

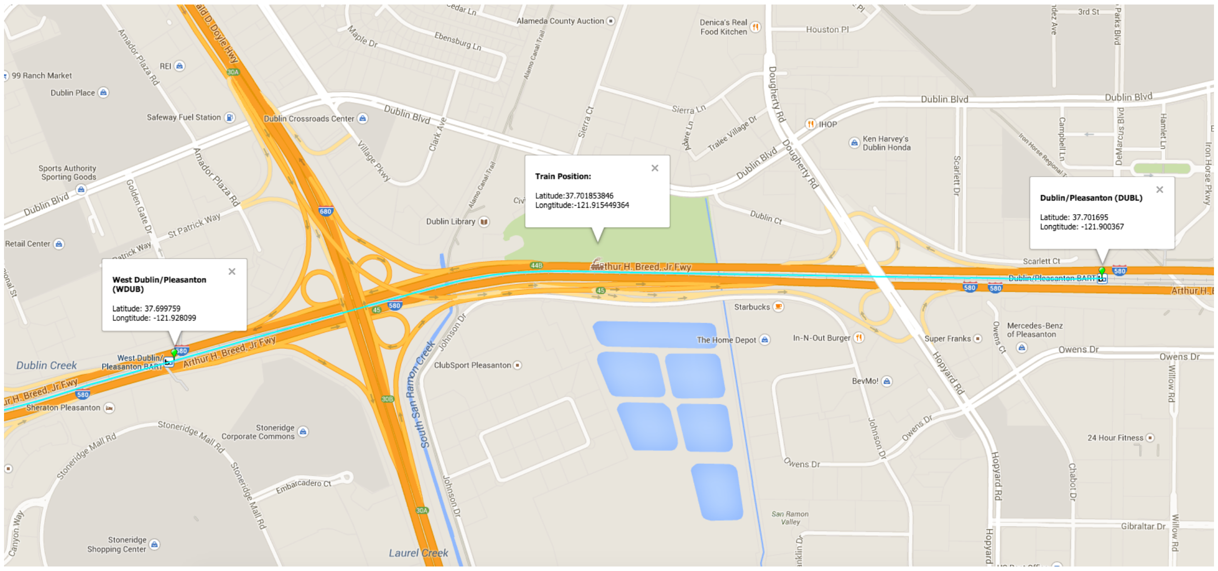
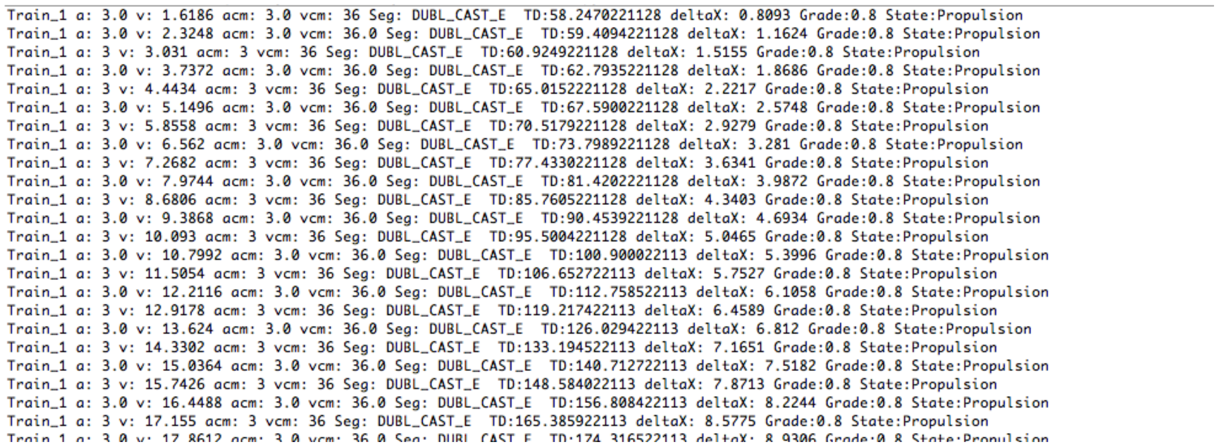


Figure 4.8: Simulation of BART train control system



```
Train_1 a: 3.0 v: 1.6186 acm: 3.0 vcm: 36 Seg: DUBL_CAST_E  TD:58.2470221128 deltaX: 0.8093 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 2.3248 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:59.4094221128 deltaX: 1.1624 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 3.031 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:60.9249221128 deltaX: 1.5155 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 3.7372 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:62.7935221128 deltaX: 1.8686 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 4.4434 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:65.0152221128 deltaX: 2.2217 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 5.1496 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:67.5900221128 deltaX: 2.5748 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 5.8558 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:70.5179221128 deltaX: 2.9279 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 6.562 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:73.7989221128 deltaX: 3.281 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 7.2682 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:77.4330221128 deltaX: 3.6341 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 7.9744 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:81.4202221128 deltaX: 3.9872 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 8.6806 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:85.7605221128 deltaX: 4.3403 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 9.3868 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:90.4539221128 deltaX: 4.6934 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 10.093 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:95.5004221128 deltaX: 5.0465 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 10.7992 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:100.900022113 deltaX: 5.3996 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 11.5054 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:106.652722113 deltaX: 5.7527 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 12.2116 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:112.758522113 deltaX: 6.1058 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 12.9178 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:119.217422113 deltaX: 6.4589 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 13.624 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:126.029422113 deltaX: 6.812 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 14.3302 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:133.194522113 deltaX: 7.1651 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 15.0364 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:140.712722113 deltaX: 7.5182 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 15.7426 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:148.584022113 deltaX: 7.8713 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 16.4488 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:156.808422113 deltaX: 8.2244 Grade:0.8 State:Propulsion
Train_1 a: 3 v: 17.155 acm: 3 vcm: 36 Seg: DUBL_CAST_E  TD:165.385922113 deltaX: 8.5775 Grade:0.8 State:Propulsion
Train_1 a: 3.0 v: 17.8612 acm: 3.0 vcm: 36.0 Seg: DUBL_CAST_E  TD:174.316522113 deltaX: 8.9306 Grade:0.8 State:Propulsion
```

Figure 4.9: Calculation result from *ComputerStation* algorithm implemented in the FRDM board

As an example, we set the initial position of the train at 50 meters from the beginning of the track. The next stop was the $WDUB$ gate located at 2447.7 meters from the beginning of the track. Figure 4.8 illustrates the location of the train on Google map. As mentioned, the train starts at 50 meters away from the beginning of the track aiming to have a full stop at $WestDublin(WDUB)$ located at [37.699, -121.928]. The picture was captured while the train was in the middle of it's way. Figure 4.8 illustrates the received data from the FRDM-K64F board which is running the algorithm. As it can be seen, in the first step of the simulation, the commanded velocity(vcm) is 36Kmph and commanded acceleration (acm) is 3Kmphps. On the other hand, the train is trying to reach the commanded velocity. The output log shows the current segment (Seg), traveled distance (TD) from the beginning of simulation, the traveled distance from the last simulation step ($deltaX$), the segment grade ($Grade$) and the train state which can be either in $Propulsion$, $Normal$, $Brake$ and $Stop$.

After running the simulation long enough, the train stopped at the desired Station. Figure 4.10, 4.11 and 4.12 illustrates the position , acceleration and velocity of the train.

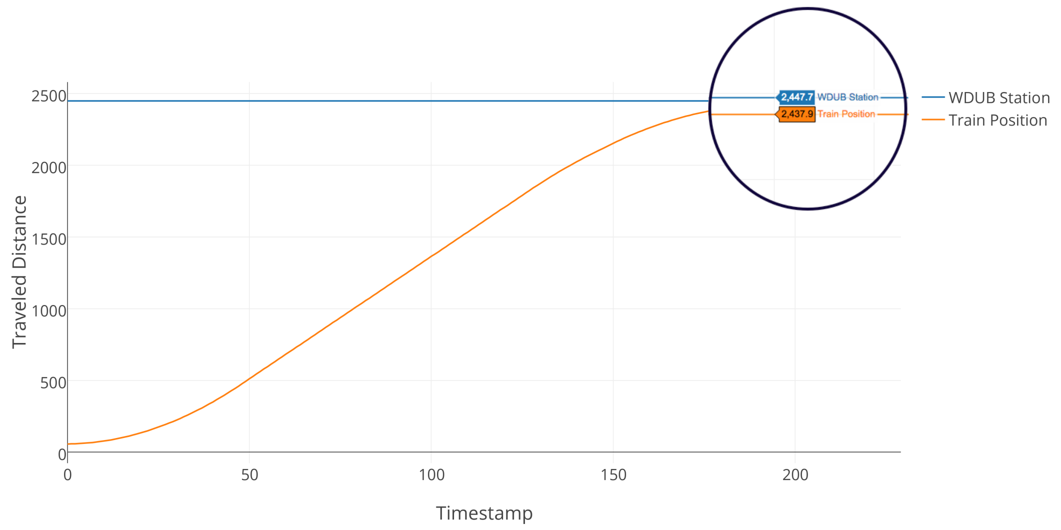


Figure 4.10: Train Position over time

As it is shown, the train obey all the safety rules defined earlier. The goal was a full stop with the range of WCSD at the $WDUB$ gate located at 2447.7 meters from the beginning of the track. Figure 4.10 illustrates that the train has successfully stopped before reaching the gate. Figure 4.11 shows that the train has always obeyed the train acceleration limit. Also, Figure 4.12 shows that the control system successfully controlled the velocity of the train as it is not reaching the maximum allowable velocity(in this case 36Kmph).

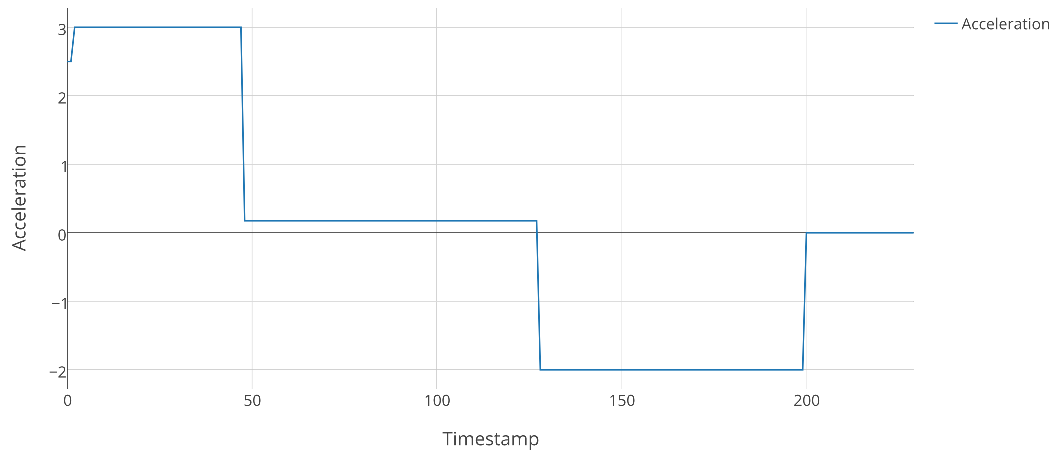


Figure 4.11: Train acceleration over time

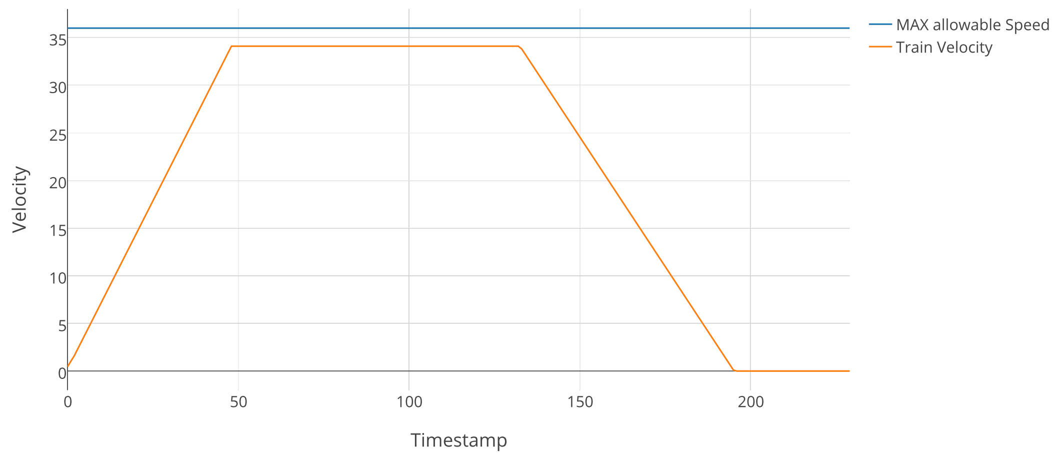


Figure 4.12: Train Velocity over time

4.2 Thread Management in JPEG Encoder

To challenge our developed tool even more, we used it to implement a JPEG encoder on the FRDM-K64 platform. In this case study, designing the whole system was not the main goal. Yet scheduling the different modules within a JPEG encoder was the main challenge. This example will demonstrate that by using our tool, how effective yet easy can it be to manage the thread execution in RTX.

A simple JPEG encoder module contains five different modules as follow:

- ***ImageReader***: This module is responsible to read the bitmap file taken from the camera and packetize it so it can be fed to the other modules.
- ***Discrete cosine transform(DCT)***: DCT [46] module in JPEG encoder is responsible to discard the low and high frequencies in the picture. There, the two-dimensional DCT-II of N $\times$ N blocks are computed and the results are quantized and entropy coded. In this case, N is typically 8 and the DCT-II formula is applied to each row and column of the block. The result is an 8 $\times$ 8 transform coefficient array in which the (0,0) element (top-left) is the DC (zero-frequency) component and entries with increasing vertical and horizontal index values represent higher vertical and horizontal spatial frequencies [46].
- ***Quantizer***: Due to the fact that the human eye is not so good at distinguishing the exact strength of a high frequency brightness variation, the amount of information in the high frequency components can be reduced. This is done by simply dividing each component in the frequency domain by a constant for that component, and then rounding to the nearest integer. If the DCT computation is performed with sufficiently high precision, the loss of the data would be negligible. As a result of this, it is typically the case that many of the higher frequency components are rounded to zero, and many of the rest become small positive or negative numbers, which take many fewer bits to represent [48].
- ***ZigZag***: This module is responsible to rearrange the components of the image in a *ZigZag* order(Figure 4.13). This is achieved by employing run-length encoding (RLE) algorithm [49] that groups similar frequencies together so that it can be used by the next module(Huffman coder)
- ***Huffman Coding***: Finally the output of the previous modules are fed into Huffman Coding module. This coding is considered as a loss-less coding meaning no part of the information will be lost in process of coding. A Huffman code is A Method for

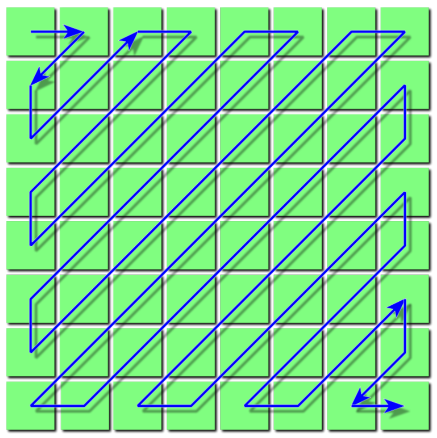


Figure 4.13: Zigzag ordering of JPEG image components[47]

the constructing a minimum-redundancy code. The method's output can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). Huffman's algorithm derives this table based on the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol

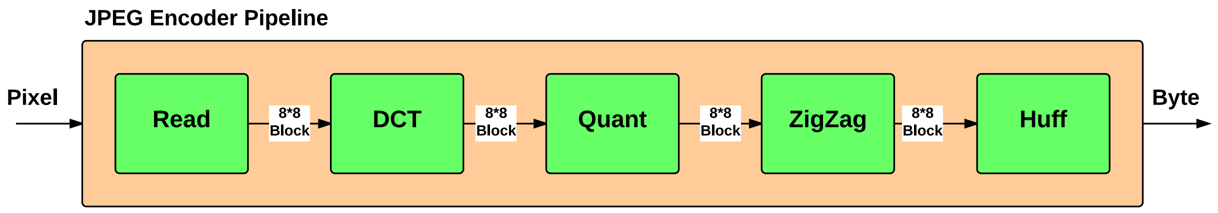


Figure 4.14: JPEG Encoder pipeline

As mentioned earlier, the goal of this case-study is not to implement the above modules. The goal is to use a high level model,such as activity diagram, to construct a high level representation of a JPEG encoder and then by employing our tool, create the implementation code from that. For this case study, we are assuming that a C code for each individual block is provided and only the scheduling is needed to be done.

Figure4.14 illustrates the data pipeline in a simple JPEG Encoder. Our goal is simply to implement this pipeline in an ARM CortexM4 device.

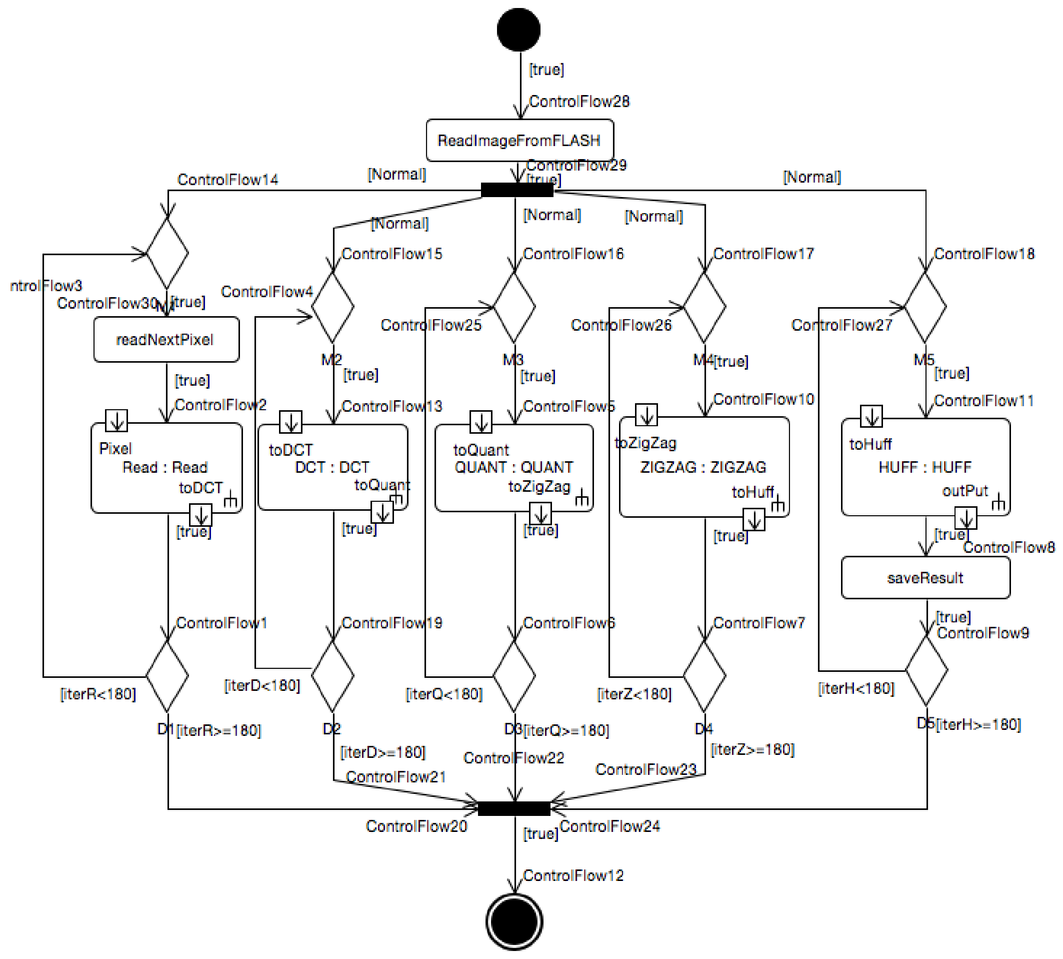
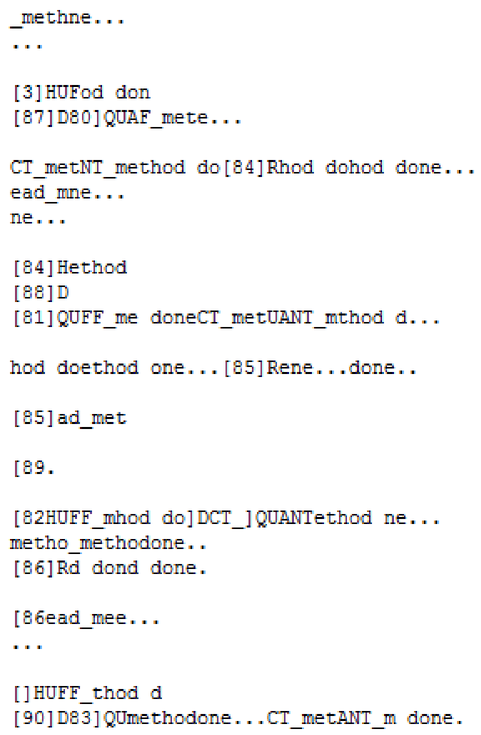


Figure 4.15: Not scheduled JPEG encoder

Figure 4.15 illustrates a high level model of a JPEG encoder. At the beginning, the image will be read from the flash memory. Then five threads will be spawned namely: $ControlFlow14$, $ControlFlow15$, $ControlFlow16$, $ControlFlow17$ and $ControlFlow18$. The priority for these threads are set as *NORMAL*. Making all these threads to have same priority is not the best way of implementing this algorithm. Later in this section, a better solution will be presented. In the reader function, the input bitmap picture is broken into 180 individual blocks. So the JPEG Encoder algorithm needs to be applied on every block. To represent this functionality, the blocks are surrounded by *Decision* and *Merge* nodes. This will create a loop behavior which will be finished after 180 iterations. The input and output of each block is demonstrated in the diagram. When the 180 iterations is finished, the program will wait until the execution of all threads is done. This is done by joining all threads. This was a non-scheduled implementation of a JPEG encoder. The reason behind non-schedule is that all the threads are spawned at the same time.



```
_methne...
...

[3]HUFod don
[87]D80]QUAF_mete...

CT_metNT_method do[84]Rhod dohod done...
ead_mne...
ne...

[84]Hethod
[88]D
[81]QUFF_me doneCT_metUANT_mthod d...

hod doethod one...[85]Rene...done..

[85]ad_met

[89.

[82HUFF_mhod do]DCT_]QUANTethod ne...
metho_methodone..
[86]Rd dond done.

[86ead_mee...
...

[]HUFF_thod d
[90]D83]QUmethodone...CT_metANT_m done.
```

Figure 4.16: Not scheduled JPEG encoder output log file

On the other hand the priority of all threads are *Normal*. This way of implementation, makes the RTX switch tasks in a very fast rate. For each iteration, we print out a notification message made after execution of the thread. After running above implementation, the log in Figure 4.16 was made and the generated JPEG pictures was corrupted. This output log shows that the thread execution was not scheduled correctly. This is due to the fact that all threads were always in READY state and only the execution timeout on the threads was causing a switch between threads. A better way of implementing this algorithm would be to schedule the execution of these threads.

Figure 4.17 illustrates a scheduled JPEG encoder. The reason is that each thread is suspended until a notification from the other module is received. Figure 4.18 illustrates the

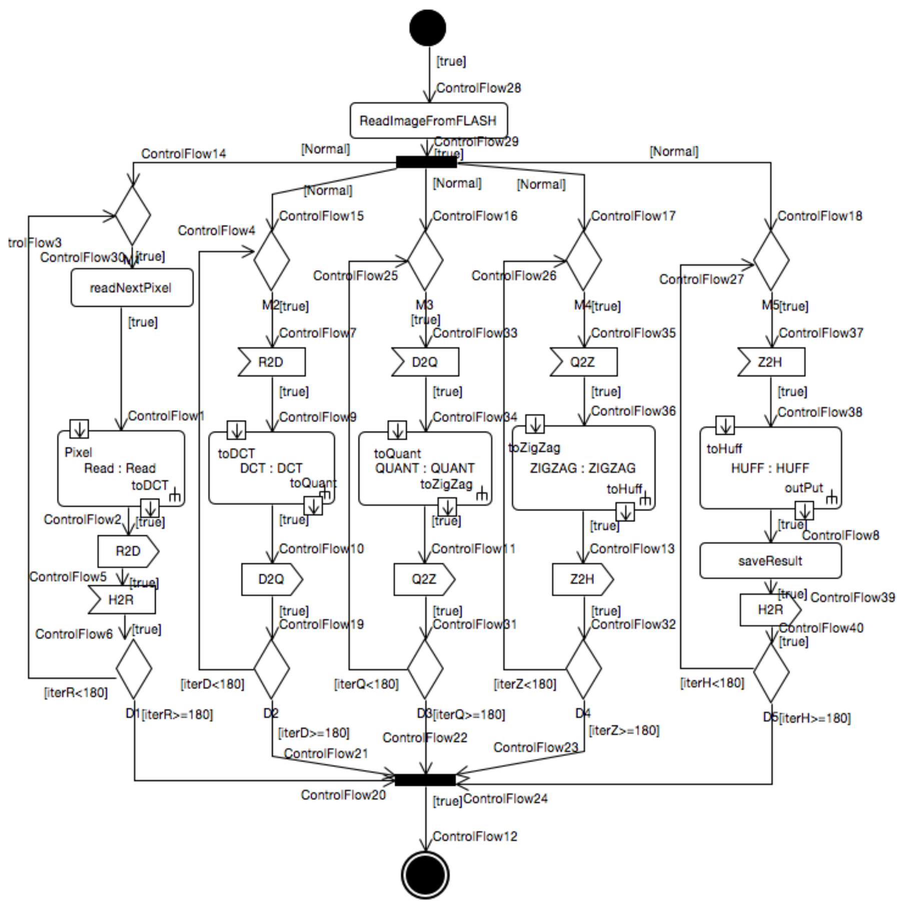


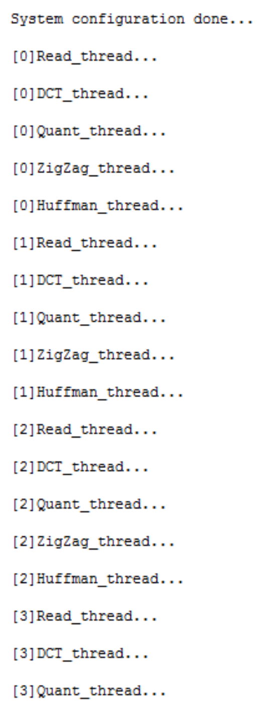
Figure 4.17: A scheduled JPEG encoder algorithm

output log after applying the simple scheduling mechanism. As it can be seen, the execution of threads is now as it was expected. The number on the left of each line shows that in which iteration the corresponding thread is.

This is not the best way of scheduling the execution of threads, but it demonstrate that,by using our developed tool, how easy it is to schedule the execution of threads in the design and then simply generate a implementation code.

4.3 Summary

In this section, two real world application was implemented in an ARM CortexM4 platform. The first application was Bay Area Rapid Transit train controller system. The train



```
System configuration done...
[0]Read_thread...
[0]DCT_thread...
[0]Quant_thread...
[0]ZigZag_thread...
[0]Huffman_thread...
[1]Read_thread...
[1]DCT_thread...
[1]Quant_thread...
[1]ZigZag_thread...
[1]Huffman_thread...
[2]Read_thread...
[2]DCT_thread...
[2]Quant_thread...
[2]ZigZag_thread...
[2]Huffman_thread...
[3]Read_thread...
[3]DCT_thread...
[3]Quant_thread...
```

Figure 4.18: A scheduled JPEG encoder algorithm Output log

controller algorithm was presented in activity diagram format. Then, our developed mapping tool was used to generate an executable code. The generated code was compiled and uploaded to the target platform. The output result confirmed the correct implementation of the algorithm.

For the second experimental application, a JPEG encoder algorithm was modeled in activity diagram format. The main modules in a simple JPEG encoder were presented as threads. A simple thread management was proposed to correct the execution of threads. Finally, by using the developed tool, the activity diagram model was mapped to the Keil RTX codes.

Chapter 5

Conclusion and Future work

5.1 Conclusion

The complexity in embedded systems has been increased in the last years. To overcome the system complexity various methodologies have been presented. Both in industry and academia, Model-Based design seems to be the best solution to solve this problem. SysML/UML diagrams are one the most popular languages in most Model-Based design tools.
In this thesis, our main goal was to generate an executable C code from a SysML/UML activity diagram models. Particularly, we were interested to automatically generate code for ARM CortexM processor family[10] from a SysML/UML activity diagram model. To achieve this goal, we proposed a set of mapping rules that were used in mapping a SysML/UML activity diagram into a suitable code to be executed on ARM CortexM processor family. To automate the process of code generation, we presented a JAVA application that used the proposed rules to automatically generate the RTX code from the input activity diagram model.
We demonstrated the capabilities of our tool by implementing two real life application. The first application was Bay Area Rapid Transit train controller system and then our tool was used to generate the RTX code.
For the second experimental application, a JPEG encoder algorithm was modeled in activity

diagram format. The main modules in a simple JPEG encoder were presented as threads. A simple thread management was proposed to correct the execution of threads. Finally, by using the developed tool, the activity diagram model was mapped to the Keil RTX codes.

5.2 Future Work

Diverse future work directions can be performed building upon this work. Here are some suggestions to improve the existing tool:

- Our current tool is mostly compatible with UML activity diagrams standard. The UML activity diagram standard is much more suitable for Software designs than system designs. This limitation affects our work when we wanted to propose mapping rules for RTX inter processes features such as mailboxes and semaphores. A very good practice would to propose a SysML package that contain all the features of Keil RTX real-time operating system. To support this feature in our existing methodology, some new NuAC terms needs to be proposed to support more features in Keil RTX real-time operating system.

- The other direction that this thesis can take was proposed earlier in the verification and implementation framework figure. Currently, the model verification is done separately on other tools(for instance AF3). To verify the model, it needs to be translated to the other verification languages. In case of AF3, the model needs to be in compliant with AF3 language. This make the verification very time consuming and inefficient. A better approach would be translating the activity diagrams to other input model-checkers language (like NuSMV). The process of translating an activity diagram to a PRISM model has been already proposed in [24] but the proposed framework did not considered an implementation option. Also, the PRISM model checker is know to be a model-checker for non-deterministic models which most likely are not suitable for implementation.

Appendix A

BART train controller code

Listing A.1: Generated Keil RTX Code

```
#include <stdbool.h>
#include "cmsis_os.h"
#include "ControlAlgorithm.h"
#include "SendCommands.h"
#include "getNewPosition.h"
#include "BART.h"

//Methods Prototype:
void InitialNode1_method(void);
void ActivityFinalNode1_method(void);
void ControlFlow1_method(void);
void ControlFlow4_method(void);
void ControlFlow6_method(void);

// Variable  Decleration :
 static  bool  InitialNode1_var  = true;
 static  bool  MergeNode1_var = false;
 static  bool  ControlAlgorithm_var  =  false ;
 static  bool  ActivityFinalNode1_var  =  false ;
 static  bool  DecisionNode1_var =  false ;
 static  bool  ControlFlow1_var  =  false ;
 static  bool  ControlFlow4_var  =  false ;
 static  bool  ControlFlow6_var  =  false ;
 static  bool  SendCommands_var = false;
 static  bool  SendCommandsEvt_var = false;
 static  char  *Data;
 static   float  acm;
 static   float  vcm;
 static   float   position ;

// Thread  prototypes :
void  ControlFlow1_thread(void  const  *arg);
osThreadDef (ControlFlow1_thread, osPriorityHigh ,  1,  0);
void  ControlFlow4_thread(void  const  *arg);
osThreadDef (ControlFlow4_thread, osPriorityNormal ,  1,  0);
void  ControlFlow6_thread(void  const  *arg);
osThreadDef (ControlFlow6_thread, osPriorityHigh ,  1,  0);

// Thread ID:
osThreadId  ControlFlow1_threadID,ControlFlow4_threadID,ControlFlow6_threadID;

// Signals :
 int32_t   SendCommandsEvt=0x1;

//  Main program  starts   here :
int  main (void){
            osKernelInitialize    () ;  //   setup   kernel
           //   Create  OS  status   variable :
```

```
        osStatus   status ;
        //  Calling  Root
        ControlFlow1_threadID  =
            osThreadCreate(osThread(ControlFlow1_thread ) ,NULL);
        //  Start  RTX kernel
        osKernelStart  () ;
}

// Methods Decleration :
void  InitialNode1_method () {
        MergeNode1_var = true;
        ControlFlow1_method(); // Calling  the  main Edge method
 }

void  ActivityFinalNode1_method() {
        print (”Program ended  successfully  ”) ;
}

// Thread  Declaration :
void  ControlFlow1_thread ( void  const  ∗arg) {
 if  (MergeNode1_var){
  if  ( InitialNode1_var  || DecisionNode1_var){
        do{ //Loop: DecisionNode1−−−>MergeNode1
            ControlFlow4_var  =  true ;
                ControlFlow4_threadID  = osThreadCreate
                    (osThread(ControlFlow4_thread ) ,  NULL);
                ControlFlow6_var  =  true ;
                ControlFlow6_threadID  = osThreadCreate
                    (osThread(ControlFlow6_thread ) ,  NULL);
                while (!( ControlFlow4_var || ControlFlow6_var)){//  Join
                    ControlFlow4_thread  and  ControlFlow6_thread
                }
                ControlFlow4_var  =  false ;
                ControlFlow6_var  =  false ;
        getNewPosition(&Data,&position);
                }while(!strcmp(Data,"Finish")) // End of  Loop: DecisionNode1 to
                    MergeNode1
                DecisionNode1_var =  true ;
                ActivityFinalNode1_method() ; //  Calling  next  node to  be  executed
        }
  }
}

void  ControlFlow4_thread(void  const  *arg){
        if  (ControlFlow4_var){
                osSignalWait  (SendCommandsEvt,osWaitForever);
                SendCommands_var = true;
                SendCommands_method(&vcm,&acm);// Calling next node to be executed
        }
        ControlFlow4_var=  true ;
}

void  ControlFlow6_thread(void  const  *arg){
        if  (ControlFlow6_var){
                ControlAlgorithm_method(&vcm,&acm,&position);// Calling  next  node
                    to  be  executed
                osSignalSet  (ControlFlow4_threadID,  SendCommandsEvt);
                SendCommandsEvt_var = true;
        }
        ControlFlow6_var=  true ;
}
```

Bibliography

[1] The International Council on Systems Engineering (INCOSE),http://www.incose.org/ , February 2015.

[2] Friedenthal, Sanford, Greigo, Regina, and Mark Sampson, INCOSE MBSE Roadmap, in INCOSE Model Based Systems Engineering (MBSE) Workshop Outbrief (Presentation Slides), presented at INCOSE International Workshop 2008, Albuquerque, NM, pg. 6, Jan. 26, 2008

[3] C. Brooks, C. Cheng, T. Feng, E.A. Lee, and R. von Hanxleden, Model Engineering Using Multimodeling, 1st InternationalWorkshop on Model Co-Evolution and Consistency Management (MCCM 08), September 2008.

[4] Object Management Group (OMG) Systems Modelling Language, http://www.omgsysml.org/, February 2015.

[5] OMG, OMG Systems Modeling Language (OMG SysML) Specification, Object Management Group, September 2007, OMG Available Specification.

[6] OMG Unified Modeling Language: Superstructure 2.1.2, Object

[7] Sysml Forum. SysML tools. http://www.sysmlforum.com/tools. htm, February 2015.

[8] Topcased the Open-Source toolkit for critical systems, http://www.topcased.org, February 2015.

[9] O. SysML. Systems Modeling Language (SysML) Specifi- cation final report. Object Management Group, 2007.

[10] ARM Cortex-M processor family, http://www.arm.com/products/processors/cortex-m/, February 2015.

[11] $Keil^{TM}$ RTX RTOS kernel. http://www.keil.com/rl-arm/kernel.asp , February 2015.

[12] $Keil^{TM}$ CMSIS RTOS. http://www.keil.com/pack/doc/cmsis/RTX/html/index.html , February 2015.

[13] ARM CMSIS standard. http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php , February 2015.

[14] μVision IDE, debugger, and simulation environment, http://www.keil.com/arm/mdk.asp, February 2015.

[15] L. Zhang, M. Glab, N. Ballmann and J. Teich, Bridging Algorithm and ESL Design: Matlab/Simulink Model Transformation and Validation, Forum on Specification & Design Languages (FDL), 2013.

[16] A. Adamov, K. Mostovaya, I. Syzonenko, A. Melnik, Electronic System Level Models for Functional Verification of System on- Chip, CADSM2007, February 20-24, 2007, Polyana, UKRAINE .

[17] M. M. R. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago and S. Olivieri, A Framework for Modeling, Simulation and Automatic Code Generation of Sensor Network Application, 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, 2008.

[18] Simulink Coder from MATHWORK,http://www.mathworks.com/ products/simulink-coder/, February 2015.

[19] Embedded Coder from MATHWORK. http://www.mathworks.com/products/embedded-coder/index.html, February 2015.

[20] F. Holzl and M.Feilkas, AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems, Model-based Engineering Of Embedded Real-Time Systems, Lecture Notes in Computer Science

[21] R. Nikhil and K. Czeck, BSV by Example. CreateSpace Independent Publishing Platform, 2010. Available: http://books.google.ca/books?id=EUmmuAAACAAJ

[22] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In CAV, LNCS, pages 585591. Springer, 2011.

[23] A. Cimatti, E. Clarke, F.Giunchiglia and M.Roveri,NUSMV: a new Symbolic Model Verifier,International Journal on Software Tools for Technology Transfer, 2000, Volume 2, Number 4, Page 410

[24] S. Ouchani, ”A Security Verification Framework for SysML Activity Diagrams”, Phd Tesis, September 2013, Concordia University, Montreal, Qc

[25] S.Ouchani, O.A.Mohamed, M. Debbabi,”A Formal Verification Framework for Blue-Spec System Verilog”, FDL, 2013

[26] G. Booch. Object-Oriented Analysis and Design with Applications (2nd Edition). Addison Wesley Longman Publishing Co., Inc., Amsterdam, 2007.

[27] J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, ”Object-Oriented Modeling and Design”, Prentice Hall, 1991.

[28] I. Jacobson, M. Christerson, and P. Jonsson. Object-Oriented Software Engineering. Addison-Wesley Professional, 1992.

[29] The taxonomy of UML diagrams, http://www.uml-diagrams.org/uml-25-diagrams.html, February 2015.

[30] M. Hause, ”The SysML Modelling Language”, Fifth European Systems Engineering Conference,18-20 September 2006.

[31] T. Weilkiens. Systems Engineering with SysML/UML: Modeling, Analysis, Design. Morgan Kaufmann Publishers Inc., 2008.

[32] Relationship between SysML and UML, http://www.omgsysml.org/, February 2015.

[33] Real Time Engineers Ltd. The FreeRTOS Project Version 8.2.0. http://freertos.org, February 2015.

[34] RTX Kernel, Theory of operation, http://www.keil.com/pack/doc/cmsis_rtx/_theory.html, February 2015.

[35] Mourad Debbabi, Fawzi Hassane, Yosr Jarraya, Andrei Soeanu, and Luay Alawneh. Verication and Validation in Systems Engineering - Assessing UML / SysML Design Models. Springer, 2010.

[36] Kordon, Fabrice, and Michel Lemoine. Formal Methods For Embedded Distributed Systems. Boston: Kluwer Academic, 2004. Print.

[37] Bart Geo-spatial Data, http://www.bart.gov/schedules/developers/geo, February 2015.

[38] AutoFOCUS3 Website. http://af3.fortiss.org/whatn isn AF3.html, February 2015.

[39] A.Campetelli,F.Holzl and P. Neubeck, User-friendly Model Checking Integration in Model-based Development, Proceedings of International Conference on Computer Applications in Industry and Engineering

[40] F.Holzl. The AutoFocus 3 C0 Code Generator Technical Report TUMI0918, Technische University at Munchen, 2009.

[41] Creating a Code Specification for a Component in AF3, http://download.fortiss.org/public/projects/af3/help/code_specification.html, February 2015.

[42] The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, http://www.omgmarte.org/, April 2015.

[43] UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, OMG Document Number: formal/2011-06-02, June 2011.

[44] BART system modeled in AF3, https://www.dropbox.com/s/i4bmrdk3rkm5ku7/AF3-Project.af3_23?dl=0

[45] FRDM-K64F: Freescale Freedom Development Platform for Kinetis. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=FRDM-K64F, February 2015.

[46] S. A. Khayam, ”The Discrete Cosine Transform (DCT): Theory and Application”, Michigan State University, March 10 2003

[47] Discrete cosine transform , http://en.wikipedia.org/wiki/Discrete_cosine_transform#JPEG, February 2015.

[48] Quantization in JPEG encoders,http://en.wikipedia.org/wiki/JPEG, February 2015.

[49] Run-Length Encoding (RLE) algorithm, http://en.wikipedia.org/wiki/Run-length_encoding, February 2015.