MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Design and Implementation of a C Code Generator Module for the Gamma Statechart Composition Framework

BSc Project Laboratory

Author
Nagy Levente Márk

Advisor
Bence Graics

June 9, 2023

Contents

Abstract

This paper presents the design and practical implementation of a C code generator module for the Gamma Statechart Composition Framework. The module serves as a vital component in the development of embedded systems software, offering seamless generation of C code from statecharts and compositions of statecharts. By leveraging the existing low-level statechart descriptors, particularly the XSTS language, the code generator module significantly simplifies the process of transforming high-level statechart models into executable C code, simplifying the process of developing embedded systems software. The resulting code is designed to be instantly compilable and can support a diverse range of hardware architectures.

Kivonat

Ez a dolgozat bemutatja a Gamma Keretrendszer számára fejlesztett C kódgenerátor modulom tervezését és gyakorlati megvalósítását. A modul fontos szerepet játszik a beágyazott rendszerek fejlesztésében, mivel lehetővé teszi C kód generálását állapotgépekből és ezek kompozícióiból. A már meglévő alacsony szintű állapotgép-leírókat, különösen az XSTS nyelvet kihasználva a kódgenerátor modul jelentősen leegyszerűsíti a magas szintű modellek transzformációját kész C kódra. A végeredmény gond nélkül fordítható a támogatott platformokon.

Chapter 1

Introduction

Code generation is becoming an increasingly popular approach for developing embedded systems software, particularly in the context of critical systems where safety and reliability are paramount. While it is crucial to design and verify embedded systems to ensure the aforementioned parameters, it is not enough; bugs and errors can still find their way into the code during implementation, making code generation an important tool for reducing human error.

This paper introduces a C code generator module for the Gamma Statechart Composition Framework designed to solve the issue of potential mistakes during implementation by eliminating the need for manual coding. Moreover, the generated code will always be consistent with the statecharts used for its creation, ensuring that the resulting software functions as intended and is easier to maintain.

Code generators are a fascinating technology due to the fact that they enable developers to separate the logical design of a software system from its implementation, therefore enabling platform-independent solutions that can be easily adapted to a variety of hardware architectures.

In conclusion, the C code generator module presented in this paper is aimed to minimize the occurrence of human errors during the implementation process, possibly even eliminating them altogether. With the ability to compile instantly on a wide variety of hardware architectures, developers across different domains can use the C code generator module to generate code that meets their specific needs, whether it be for aerospace, automotive, medical devices, or any other domain where reliable software is a must.

Chapter 2

Background

The second chapter provides the contextual information needed to comprehend the work presented. It commences by introducing model-driven development, specifically statecharts, in Section 2.1. The Gamma Statechart Composition Framework, a modeling tool employed in this paper, is then described in Section 2.2. The chapter also describes the XSTS Language used by the code generator (Section 2.3), and how it corresponds to the C language. The chapter concludes with Section 2.4, which presents modeling tools and approaches related to the work.

2.1 Introduction to Model-Driven Development

Model-driven development [4] (MDD) (or model-driven software engineering) is a software engineering methodology that draws attention to the use of models to represent the requirements, design, and implementation of software systems. Instead of writing code directly, developers create high-level models that describe the system's behavior and structure, and then use tools to generate the code automatically.

In this paper, we will discuss statecharts, a modeling technique used in model-driven development. Statecharts are visual representations of state machines [6], which provide a high-level abstraction of a system's behavior.

Definition. The mathematical definition of state machine M can be M = $(\Sigma, S, s_0, \delta, F)$.

- Σ is the input alphabet (a finite non-empty set of symbols)
- S is a finite non-empty set of states
- $s_0 \in S$ is the initial state
- δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$ (in a nondeterministic finite automaton it would be $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$, i.e. δ would return a set of states)
- F is the set of final states, a (possibly empty) subset of S

2.2 Introduction to Gamma

The Gamma Statechart Composition Framework [11] is an Eclipse-based, integrated modeling and analysis toolset for the component-based design of reactive systems. The framework intentionally reuses statechart models of existing tools and their respective code generators for individual components. As a core functionality, it provides a composition language that supports the interconnection of statechart components in a hierarchical way. In addition to modeling, Gamma provides automated code generators for both atomic statechart and composite models and also support system-level formal verification and validation by mapping statechart and composition models into formal models of various model checkers and back-annotating the results. The framework also provides test generation functionalities for the interactions between the components using the integrated model checker back-ends [10].

This work centers on enhancing Gamma's capabilities by enabling the generation of C code from statecharts and compositions of statecharts.

2.3 Gamma Compositions

The Gamma framework supports synchronous and asynchronous compositions to define the behavior of components. For code generation, only synchronous compositions are being used.

2.3.1 Synchronous

The synchronous-reactive composition defines that the components run in parallel, their inputs are read at the beginning of the cycle, so there is no communication within the cycle. The cascade composition defines that the components run in discrete times, one after the other - by default in the order of declaration. Their inputs are read just before execution, so communication within the cycle is possible. A component can be executed multiple times within one cycle.

2.3.2 Asynchronous

The asynchronous-reactive composition defines that the components run in parallel, the communication goes through message queues, which behave as a FIFO data structure, but a priority-based solution is also possible. If the storage is full, any further incoming message is ignored.

2.4 Related Tools

2.4.1 Yakindu Statechart Tools

For statecharts, we use Yakindu [8], which is a modeling tool that provides support for the development of reactive, event-driven systems based on the concept of state machines. The elements that Yakindu provides to represent a statechart include regions, states, transitions, events, and actions. States represent the different states that a system can be in, and transitions represent the possible transitions between those states. Events represent the triggers that cause state transitions, and actions represent the behavior that is executed when a transition occurs. Guards are used to ensure that a transition can only occur if a certain condition is met. Initial, shallow, and deep history states are special types of states in a statechart that have distinct behaviors and are used to define the initial state and the memory of the machine.

2.4.2 Eclipse EMF

Eclipse EMF [1] (Eclipse Modeling Framework) is a widely-used modeling and code generation framework for developing applications based on structured data models. It provides an infrastructure for creating and managing structured data models using a metamodel, which defines the structure and constraints of a model.

EMF is used to implement the XSTS Language, providing a collection of XSTS elements for constructing XSTS models. These elements are utilized to construct a low-level representation of statecharts or compositions of statecharts. The EMF implementation also provides support for serialization and deserialization of XSTS models in XML, as well as a framework for generating code from the models using templates.

2.5 The XSTS Language

The EXtended Symbolic Transition Systems [9] (XSTS) formalism is an intermediate representation aiming to represent reactive systems. It is an extension of the symbolic transition systems (STS) formalism, which is a low-level representation. XSTS introduces an imperative layer above the SMT formulas of STS models, making it easier to transform high-level engineering models into XSTS and map them to different tools, such as model checker back-ends and code generators.

An XSTS model starts with custom type declarations, similar to enum types in programming languages, which are generated for each region of the statechart model with their contained states as literals. It is followed by global variable declarations with integer, boolean, and custom types, including array types. The behavior of the system is defined by three atomic transitions: init, env, and trans. The init transition initializes the system, while the env transition describes the behavior of the system's environment. The trans transition describes the system's internal behavior and alternates with the env transition. The statements within each transition define the detailed behavior of the system, such as assigning values to variables, acting as guards, or creating transient variables. The language also supports composite statements that contain other statements, such as sequences, choice statements, parallel statements, if-else statements, and for loops over ranges.

Chapter 3

Modeling and Code Generation

This chapter covers various aspects related to modeling and code generation. It begins by discussing the difficulties that arise when generating code from low-level model descriptors for embedded systems. The chapter then goes on to cover the Gamma statecharts and its composition language, model transformations, and mapping XSTS to C code.

3.1 Code Generation for Embedded Systems

This section discusses the challenges that arise when generating code from low-level models. Embedded systems [2] software is required to run on various hardware architectures to ensure flexibility and portability on different hardware components, reduce constraints on physical device selection, and provide long term support. Therefore, a high-level design language is needed to model the reactive system's behavior independently of the hardware and to enable software development on multiple platforms.

UML Statecharts [7] serve best as the high-level design language for modeling component based reactive systems due to their ability to capture complex state behavior and event handling, greatly reducing complexity compared to programmer-made code. Gamma provides a composite language to build compositions of statecharts which helps to reduce complexity and increase modularity. As of now, Gamma lacks the capability to generate platform independent code apart from the already existing Java code generator which is not optimal for embedded systems due to its high resource requirements and lack of real-time support. The following objectives should be met by a code generator intended for embedded systems:

- **Limited resources:** The code should be runnable on wide-range of hardware architectures. Its choice of language should consider resource requirements and compiler support for embedded hardware.
- **Portability:** The generated code should be portable across different platforms to allow for flexibility and use among different domains, from the use of embedded Linux to programming microcontrollers.
- **Safety and reliability:** Embedded systems are often used in critical applications such as medical devices and transportation systems, therefore the generated code should be safe and reliable.

Considering the aforementioned requirements, I have opted for the C programming language as it demands minimal resources and has extensive support from compilers used for embedded hardware along with software verification tools.

3.2 Gamma Statecharts

In Gamma, components are defined using statecharts, which are based on the Unified Modeling Language (UML). Statecharts consist of states that define the state of the component as well as how it reacts to incoming events. Events are responsible for initiating a possible state change and might have parameters linked to it. Events can come from outside the component, or can be produced by internal timers. Actions are executable statements that are executed when a transition occurs, while guards are conditions that must be met for a transition to occur. Timeout events can be associated with states and transitions in statecharts, which trigger after a specified duration of time. Upon entering or exiting a state, entry and exit actions can be executed. These actions have the ability to fire events. UML Statecharts also include hierarchical states, which allow for the composition of multiple states into a higher-level state, and the ability to nest states within other states. Shallow and deep history states store the last active state of a higher-level state when it is exited. When re-entering the high-level state, the active inner state will be the one stored by the history state in case of deep history.

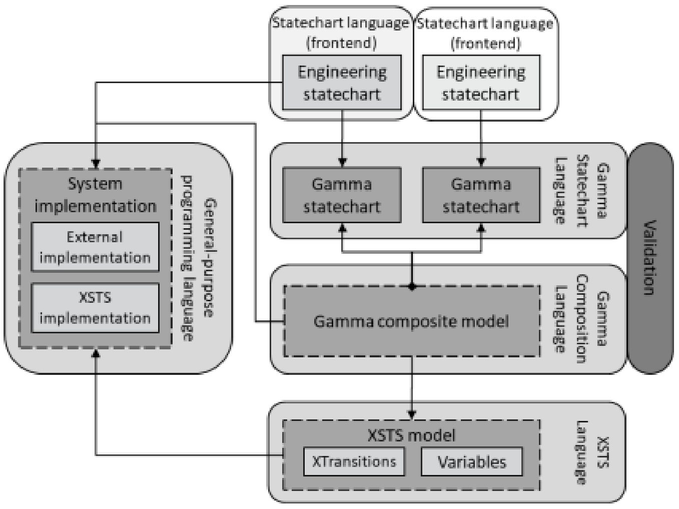

Figure 3.1: The route of models to code generation [10].

Statecharts provide a high-level representation of a system's logic. While manual implementation of this behavior is possible, it is error-prone and time-consuming. Automatic code generation can eliminate these issues, but it requires a bridge between the high-level statechart model and the low-level programming language. Gamma addresses this issue by transforming its statechart components into low-level representations, which enables developers to work with high level models, such as statecharts, while code generators can utilize the generated low-level representations.

3.3 Model Transformation

Model transformations play a crucial role in the code generation process, enabling the conversion of statechart models through various stages to achieve specific goals. The sequence begins with UML statecharts obtained from Yakindu, a popular modeling tool. These UML statecharts are being transformed into the Gamma statechart language, from which Gamma compositions can be created using the Gamma composite definition language. Gamma compositions allow for the combination of multiple statecharts to create complex systems.

To facilitate formal verification, the Gamma compositions can then be transformed into languages recognized by specific model checkers such as UPPAAL, Theta, or Spin. Another transformation step involves converting the statechart descriptors or compositions into the XSTS language. Although this intermediate step is not discussed in detail here, it serves as a bridge between Gamma languages and XSTS.

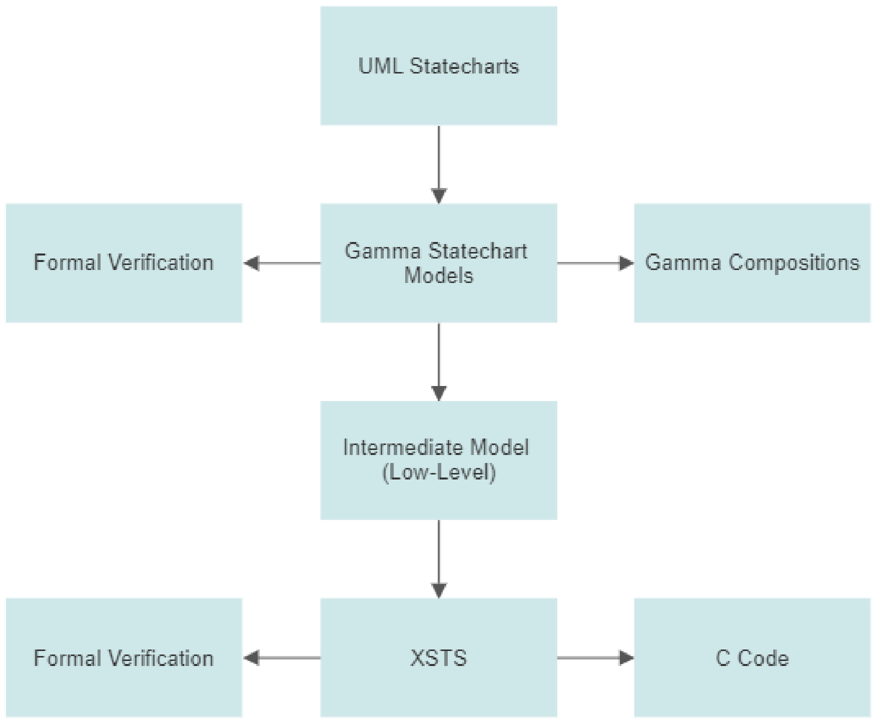

Figure 3.2: Model transformations using Gamma [12].

The XSTS models serve as the foundation for generating code in languages like C or Java. Moreover, the XSTS models can be utilized for formal verification by further transforming them into languages recognized by the model checkers mentioned earlier. This enables comprehensive formal verification of the system's properties and ensures its correctness and reliability. By acquiring these test cases in the Gamma test language, we can transform them further into test cases to not only verify our model but the code generated by the generator itself.

3.4 Mapping XSTS to C Code

Once a statechart model has been transformed to its low-level descriptor, it needs to be translated to C language to be compiled and executed on the target platform. In the context of this paper, the XSTS model is transformed into C code. This process requires mapping the elements and behavior of the XSTS language onto constructs and syntax within the C language.

Since the code generator uses a version of XSTS serialized to XML, containing specific types of generated objects by using Eclipse EMF we cannot explicitly map XSTS keywords to C expressions. However the structure of one can be easily corresponded to another.

The mapping begins with the type definitions present in the XSTS model. These type definitions are mapped to C enums. The enums serve as symbolic representations of the regions and states within the system. Following the type definitions, the variable declarations in the XSTS model are mapped to variables in the C code. These variables represent the underlying mechanism of the component. In the C code, these variables are declared within a struct. Common data types used in the variable declarations include integers (unsigned if the XSTS variable has a ClockVariable annotation), floats, booleans, and the previously mentioned enums.

```
<typeDeclarations name="Main_region_Controller">
  <type xsi:type="hu.bme.mit.gamma.expression:EnumerationTypeDefinition">
    <literals name="__Inactive__"/>
    <literals name="Operating"/>
    <literals name="Interrupted"/>
  </type>
</typeDeclarations>
```

Listing 3.1: Representation of regions in XSTS.

The XSTS model includes variable groups, specifically SystemInputGroups and SystemOutputGroups, along with their associated ParameterGroups. These groups are mapped to ports in the C code, effectively generating setters and getters for the system's inputs and outputs. Furthermore, the XSTS model contains initializing transitions, which define the initial state and behavior of the system. These transitions are consisting mainly of value definitions, setting initial values for variables within the struct.

Finally, the XSTS model includes transitions that consist of actions and expressions. These transitions capture the core logic and behavior of the system. Actions and expressions are mapped to their respective counterparts in the C code. Due to the extensive number of actions and expressions present in the XSTS model, it is impractical to individually introduce each one within the scope of this section.

Chapter 4

Implementation

This section provides an in-depth explanation of the code generator's architecture and implementation. First, it begins with the requirements of the code generator itself, then section 4.2 delves into the specifics of the results after generation. Following that, section 4.3 explores the architecture and design used within the project. Lastly, the chapter concludes with a discussion on the current limitations of the code generator, highlighting any known constraints or areas for future improvement.

4.1 Requirements

The C code generator reuse architectural and design patterns from the already existing Java code generator[11] of Gamma. Though the two languages are very different, they both have to provide the same functionalities to correctly implement a component.

Each generated implementations must be able to:

- Initialize, reset the component
- Execute a step on the component
- Reset in-, outputs between steps

The following section presents the architecture that made the aforementioned demands possible.

4.2 C Code Generation

Within the generated code, Gamma components are implemented as C structs, while regions are represented as an enums. Initializations, resets, and transitions, which define the behavior and state changes of the system, are implemented as functions.

Unlike object-oriented languages such as Java, C does not provide built-in support for interfaces, classes, or namespaces. The absence of the aforementioned features poses a challenge when attempting to apply modular code organization in C development. To overcome these limitations, the C code generator take advantage of the fact that two components cannot have the same in XSTS. While this approach works for regions and components, there is a possibility that states within different regions cause conflicts. To solve yet another problem connected to the absence of namespaces, the adoption of a specific naming convention is required. By appending the name of the region after each state having an underscore between, we have the solution we needed.

```c
/* Enum representing region Main_region_Controller */
enum Main_Region_Controller {
  __Inactive___main_region_controller,
  Operating_main_region_controller,
  Interrupted_main_region_controller
} main_region_controller;
```

Listing 4.1: Representation of the Main Region in C

The generation of necessary functions is carried out similarly to the generation of regions. The main difference between the two conventions is the appendix. In case of functions and routines, we append the name of the component to which it belongs.

```c
/* Reset component Crossroad */
void resetCrossroadStatechart(CrossroadStatechart* statechart);
/* Initialize component Crossroad */
void initializeCrossroadStatechart(CrossroadStatechart* statechart);
/* Entry event of component Crossroad */
void entryEventsCrossroadStatechart(CrossroadStatechart* statechart);
/* Clear input events of component Crossroad */
void clearInEventsCrossroadStatechart(CrossroadStatechart* statechart);
/* Clear output events of component Crossroad */
void clearOutEventsCrossroadStatechart(CrossroadStatechart* statechart);
/* Transitions of component Crossroad */
void changeStateCrossroadStatechart(CrossroadStatechart* statechart);
/* Run cycle in component Crossroad */
void runCycleCrossroadStatechart(CrossroadStatechart* statechart);
```

Listing 4.2: Function declarations in C

Introducing a wrapper statechart, as an extra layer above the component, is essential as it handles timing, abstracts internal mechanisms, and provides ports for interaction. The C language does not support encapsulation in the form of private attributes, therefore the user can interact with the component directly despite the fact that the code is meant to be hidden. Through ports, it enables controlled access to internal state and data. These ports were realized as setters / getters for the ease of use. By managing timing-related behaviors, it ensures the independence of our layers, one being the component itself which is responsible for the internal mechanism, while the responsibilities of the wrapper component were already summarized.

4.3 Structure of the Code Generator

The structure of the code generator consists of several packages and classes. In the main directory, the *IStatechartCode* interface is provided, which serves as the contract for each source and header file pair. The *CodeBuilder* class implements this interface and is responsible for constructing the source and header files for the components. Similarly, the *WrapperBuilder* class also implements the *IStatechartCode* interface and is responsible for building the wrapper source and header files.

Within the model package, there are two models and a common abstract base. The *FileModel* class is the latter that handles file related operations. The *CodeModel* class represents the model for the source code files, while the *HeaderModel* class represents the model for the header files.

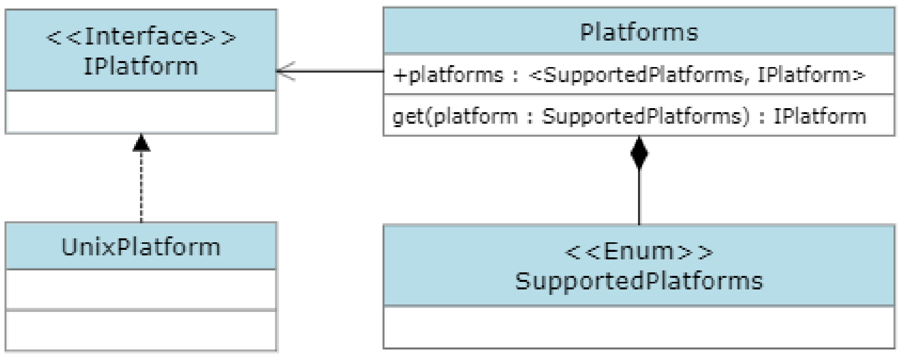
<<Interface>>
IPlatform
Platforms
+platforms : <SupportedPlatforms, IPlatform>
get(platform : SupportedPlatforms) : IPlatform
UnixPlatform
<<Enum>>
SupportedPlatforms

Figure 4.1: The structure of the platforms package.

The platforms package is specifically designed for platform-independent timing code generation. It contains the *IPlatform* interface, which is required by each platform implementation. At the end of this semester there is only one supported platform:

- Unix Platform

The serializer package plays a vital role in serializing various components of the XSTS models into C code. It includes the *ActionSerializer* class, which serializes actions into C code, the *ExpressionSerializer* class for serializing expressions, the *TypeDeclarationSerializer* class for serializing type declarations into C enums, and the *VariableDeclarationSerializer* class, which serializes variable declarations into C code, potentially within the component struct.

4.4 Current Limitations

The code generator has certain limitations that should be considered. For instance, it does not provide support for all types of gamma compositions, such as any asynchronous composition. Additionally, it is currently limited to Unix platforms, which may restrict its usage in certain environments. Moreover, the precision which we measure elapse time with also acts as a limiting factor. The model stores clock values as integers in milliseconds. Its granularity may not be sufficient for all applications that require more precise timing. Rounding errors may occur when converting between different units, leading to potential inaccuracies. Additionally, the code generator relies on variables to track elapsed time. In certain scenarios, if the elapsed time exceeds the maximum value that can be stored in the certain variable type, in our case an unsigned integer, an overflow may occur. This can lead to unexpected behavior or even segmentation faults in the generated code.

Chapter 5

Conclusion

In this chapter, we present a case study that demonstrates the practical implementation of a crossroad example using a Raspberry Pi [5]. This case study showcases the application of embedded systems and highlights the integration of hardware and software components.

5.1 Case Study

In the Gamma tutorial, a crossroad example is presented for illustrating statechart compositions, model transformations, formal verification on an initially faulty model. The tutorial has a *Controller* that synchronizes the two *TrafficLight* objects, and provides a way to interrupt the normal workflow of the crossroad. The tutorial uses inbound and outbound ports to represent the interaction points of the system with its environment, the aforementioned police interrupt button and 4 event for each lamp component that represents the 3 color (red, yellow, green) plus an extra event where no lamp is on. By applying the C code generator to this tutorial example, we can transform the statechart composition into C code that utilizes the specified inbound and outbound ports to transfer these events to a physical form, utilizing a custom hardware board.

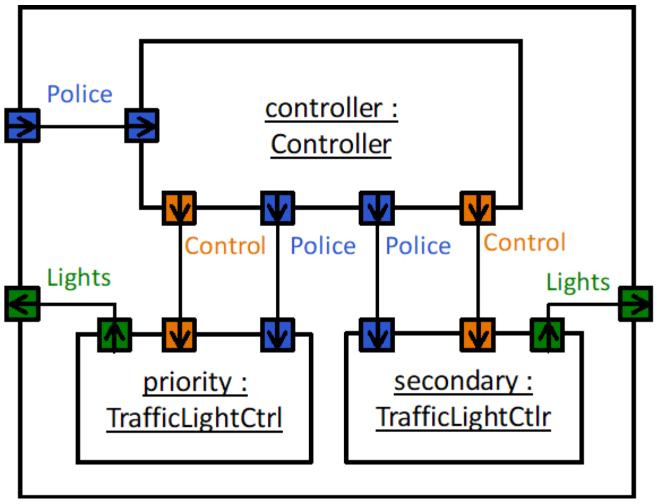

Figure 5.1: The overview of the crossroad component.

The hardware configuration, acting as a hat for the Raspberry Pi, expands its functionality to act as a crossroad system. The crossroad hardware comprises two red LEDs, two yellow

LEDs, and two green LEDs, representing the traffic lights, as well as a button that serves as police interrupt signal.

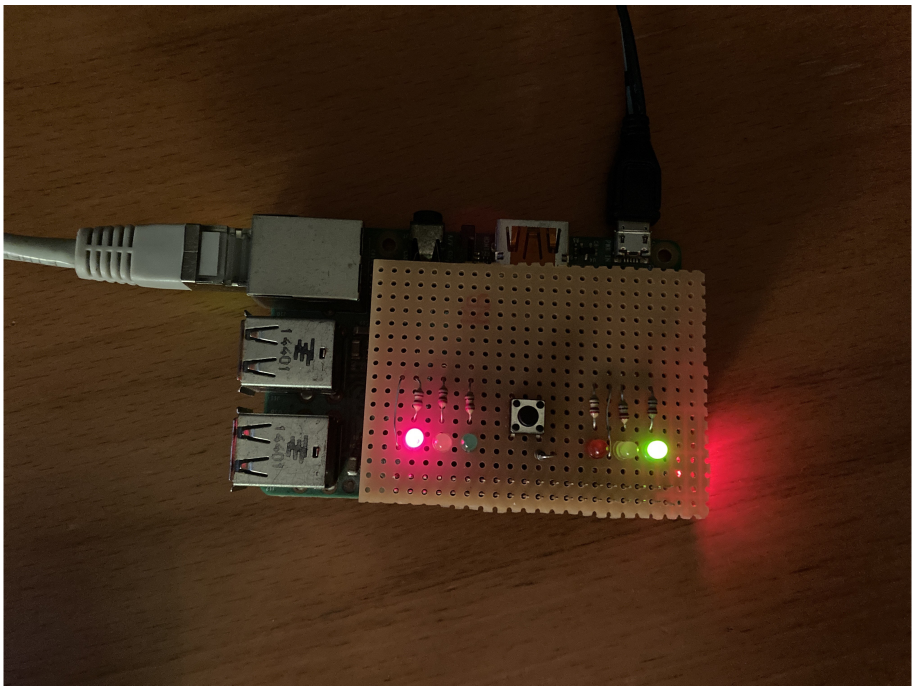

Figure 5.2: The code deployed on a Raspberry Pi.

After generating the C code from the Gamma tutorial's crossroad example, I found that the generated code worked seamlessly in controlling the simulated crossroad system. However, to fully drive the model and observe its behavior, I had to create an additional file that acted as the driver. This driver source file was responsible for reading the output events generated by the model and setting the input events, which in this case were triggered by a button. The button input was detected by monitoring its rising edges, and the corresponding output events were used to control the behavior of the LEDs. By developing this module, I was able to establish a complete interaction loop between the simulated crossroad system and its environment, enabling me to observe the desired outputs reflected through the LEDs based on the input events triggered by the button. This additional file served as a bridge between the generated code and the hardware components, ensuring the seamless integration and functionality of the model in a practical setting.

5.2 Overview

This case study provides an overview of the successful generation and compilation of code from the Gamma tutorial's crossroad example onto a Unix platform, specifically the Raspberry Pi. One of the key highlights of this case study is the ease with which the generated code was compiled without encountering any errors or warnings. This case study underscores the practicality and efficiency of using the Gamma framework and its code generator module for developing embedded systems software.

Chapter 6

Future Work

In terms of future work, there are several areas that can be explored to enhance the capabilities of the code generator. Firstly, addressing the timing-related limitations by searching for a way to measure time with sufficient granularity while keeping overflow issues above an acceptable level, potentially solving them altogether. It requires thorough planning, since in cases where this problem is neglected segmentation faults can happen, on the other hand modifying the values of clock variables may result in nondeterministic behavior, possibly triggering unwanted events.

Expanding the platform support is another important aspect. While the code generator currently targets Unix platforms, extending its compatibility to handle other operating systems, scenarios where an OS is not present would broaden the code generators applicability in general.

Yet another significant direction for future development is the generation of unit tests for the generated code. By leveraging test cases provided by model checkers, it would be possible to automatically generate unit tests that cover our model's state space for the given coverage-criteria. Integrating formal verification [3] techniques into the code generation process would provide a higher level of confidence in the generated code's behavior, possibly catching any error during generation that may result in the discovery of underlying problems within the code generator itself.

Bibliography

[1] Eclipse emf, 2023. URL `https://en.wikipedia.org/wiki/Eclipse_Modeling_Framework`.

[2] Embedded systems, 2023. URL `https://en.wikipedia.org/wiki/Embedded_system`.

[3] Formal verification, 2023. URL `https://en.wikipedia.org/wiki/Formal_verification`.

[4] Model-driven engineering, 2023. URL `https://en.wikipedia.org/wiki/Model-driven_engineering`.

[5] Raspberry pi, 2023. URL `https://www.raspberrypi.com/`.

[6] Finite state machines, 2023. URL `https://en.wikipedia.org/wiki/Finite-state_machine`.

[7] Uml statemachines, 2023. URL `https://en.wikipedia.org/wiki/UML_state_machine`.

[8] Yakindu statechart tools, 2023. URL `https://www.itemis.com/en/products/itemis-create/`.

[9] Vince Molnár István Majzik Bence Graics, Milán Mondok. Test generation and formal verification for asynchronously communicating distributed controllers, 2023.

[10] János Csanád Csuvarszki. Model-driven development of heterogeneous cyber-physical systems, 12 2020.

[11] Budapest University of Technology, Department of Measurement Economics, and Information Systems. The gamma statechart composition framework, 2023. URL `http://inf.mit.bme.hu/en/gamma`.

[12] Nyika Sándor. C++ kód generálása elosztott állapotgép modellekből, 12 2019.