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

Model-Driven Development of Heterogeneous Cyber-Physical Systems

Scientific Students' Association Report

Author:

János Csanád Csuvarszki

Advisor:

dr. András Vörös
Bence Graics

2020

Contents

Kivonat

Napjainkban egyre több iparág épít kiberfizikai rendszerekre a kritikus feladatok megbízható ellátásához. Az ilyen rendszerek jellemzője, hogy nem csak egyetlen programból, hanem több, különböző fizikai eszközön futó szoftver- és hardverkomponensből épülnek fel, ideértve egy adott környezeten belül található különböző programozható vezérlőket és a hozzá integrált szenzorokat és beavatkozókat, a manapság terjedő fog és edge számítási megoldásokon futó folyamatokat, vagy a felhőt. Emiatt ezen rendszerek heterogén rendszernek tekinthetők, amelyek tervezése és ellenőrzése rendkívül komplex feladat. A komplexitás kezelésének gyakori eszköze a modellvezérelt megközelítés, amely segítségével magas szinten leírható mind az egyedi komponensek működése, mind ezek egymással történő kommunikációja. Kellően részletes modell esetén lehetőség nyílik az implementáció automatikus származtatására is, amely felgyorsítja a fejlesztést és lehetőséget ad a hibák korai felismerésére, javítására.

A Gamma Statechart Composition Framework egy a hierarchikus, komponens-alapú reaktív rendszerek tervezését és ellenőrzését támogató eszköz. A komponensek integrációját egy formális szemantikával rendelkező kompozíciós nyelv segítségével többféle kompozíciós mód (aszinkron-reaktív, szinkron-reaktív és kaszkád) szerint támogatja, viszont ezek a kompozíciós módok jelen formájukban kevésbé alkalmasak heterogén rendszerek és az azokban előforduló interakciók leírására. Ezen felül a keretrendszer nem támogatja sem a rendszermodellekből a beágyazott rendszerekben jellemzően használt forráskód automatikus generálását, sem az egyes (heterogén) komponensek hálózaton keresztül történő kommunikációját.

Munkám során a Gamma keretrendszert bővítem új funkcionalitásokkal, melyek támogatják a heterogén rendszerek tervezését és az implementáció automatikus származtatását. Ezen rendszerek interakcióinak pontos leírásának érdekében heterogén architektúrákban alkalmazható kompozíciós szemantikai variánsokkal egészítem ki a már meglévő szemantikákat. Az egyedi szoftverkomponensek real-time vezérlőkön történő futtatására, illetve egyedi hardverkomponensek szintézisének támogatására automatizált kódgenerátorokat fejlesztek, lehetővé téve a beágyazott szoftver előállítása mellett logikai áramkörök viselkedésének leírását is. Támogatok többbféle, kritikus és valós-idejű beágyazott rendszerekben használt kommunikációs megközelítést.

A megoldásom alkalmazhatóságát két esettanulmányon szemléltetem, egyik esetben egy real-time vezértlőből és FPGA-ból álló heterogén architektúrán, másikban egy halózaton kommunikáló elosztott környezetben futtatva a rendszert.

Abstract

Nowadays, cyber-physical systems are becoming more prevalent in the industry for the reliable execution of critical tasks. As a frequent characteristic, these systems consist of not only a single program but multiple software and hardware components running on different physical devices, including programmable components in an embedded environment with their integrated sensors and actuators, processes running on fog and edge solutions, or the cloud. As a result, they can be considered heterogeneous systems whose design and analysis are remarkably complex. Complexity can be handled by introducing a model-driven approach, which aids the high-level description of both the behavior of standalone components and their communication. The definition of sufficiently detailed models allows for the automatic derivation of implementation, making development faster and allowing the detection and correction of errors in the early stages.

Gamma Statechart Composition Framework is a tool for the design and analysis of hierarchical, component-based reactive systems. The integration of components is facilitated by a composition language with formal semantics supporting multiple composition modes (asynchronous-reactive, synchronous-reactive and cascade). However, these semantics in their current form are slightly suitable for describing heterogeneous systems and their typical interactions. Furthermore, the framework does not support either the automatic derivation of source code typically used in embedded environments from system models or the communication of standalone components via network.

In this work, I extend the Gamma framework with new functionalities, which support the development of heterogeneous systems and the automatic derivation of implementation. I extend the existing composition semantics with semantic variants tailored to heterogeneous architectures for the precise description of interactions. To allow the execution of standalone software components on real-time controllers and support the synthesis of standalone hardware components, I develop automated code generators, enabling both the derivation of embedded software as well as the description of logical circuit behavior. I support multiple communication solutions used by critical, real-time embedded systems.

I present the applicability of my solution in the context of two case studies, one executed on a heterogeneous architecture consisting of a real-time controller and an FPGA, and the other realising a distributed networked communication.

Chapter 1

Introduction

As digital technology advances, we become increasingly dependant on computer devices in our everyday lives. Apart from desktop computers and smartphones, which we actively use and carry with ourselves, numerous devices (embedded systems) surround us, usually as parts of more complex systems, such as cyber-physical systems (CPS). Cyber-physical systems are deeply integrated into their environment: not only do they derive data from their surroundings, but they actively alter the physical world in order to achieve certain goals. As a result, these systems have to be configurable and fault-tolerant. A frequent characteristic of CPS is that they consist of many heterogeneous components, such as embedded devices with sensors and actuators, and can connect to processes running on fog and edge solutions, or in the cloud. This heterogeneity makes the development and analysis of cyber-physical systems a cumbersome task.

As an example, the idea of developing medical monitoring devices in the context of cyber-physical systems is becoming increasingly prevalent [3]. Such devices monitor the physiological status of a patient using sensors. The connection of these sensors allows for easier data collection, which can then be stored and compared to other data entries using big data technologies [2]. Furthermore, medical CPS also has to account for intelligent decision making based on the patients' data, which can be solved by utilising high-performance cloud solutions. Since medical a CPS can directly interfere with humans, it must be safety-critical. This example demonstrates how complex these systems can be, as the heterogeneous components of CPS have to constantly interact with each other while conforming to reliability and quality of service requirements.

This complexity is usually handled by introducing a model-driven development approach, which focuses on the high-level description of the system. This allows developers to focus on the structure, behaviour and communication of the components instead of low-level implementation details. Model-driven approaches can support verification and validation (V&V) techniques, and based on a sufficiently detailed model, automatic code generation is also feasible.

However, most modeling tools do not provide support for the design of heterogeneous systems, which would require the systematic integration of models describing the behavior of standalone components. Furthermore, the automatic derivation of implementation for different hardware components is also rarely supported. Therefore, approaches aiming to support the model-driven design of cyber-physical systems have to support *1)* the integration of components with well-defined execution and interaction semantics tailored to heterogeneous systems and *2)* automatic code generators for different software and

hardware platforms, and communication modes. As a solution, I propose an approach in the context of the Gamma Statechart Composition Framework.

The Gamma Statechart Composition Frameworks is a tool for the design and analysis of hierarchical, component-based reactive systems. The integration of components is supported by multiple composition modes (asynchronous-reactive, synchronous-reactive and cascade). However, these composition modes in their current form are slightly suitable for the design and description of heterogeneous cyber-physical systems. There is a strong need to describe data-driven communication commonly found in CPS, where the main form of communication between components is data transmission. Furthermore, the framework does not support either the automatic code generation from system models in languages typically used in embedded systems or the communication of components via network.

The goal of this work is to extend the Gamma framework with new functionalities that support the design and development of heterogeneous cyber-physical systems and the automatic generation of implementation. To achieve this,

- I extend the asynchronous semantics with a data-centric semantic variant suitable for the description of data-driven communication commonly found in heterogeneous systems;
- I develop code generators that target embedded processors and FPGAs, allowing for the execution of standalone software components on embedded controllers, and the synthesis of standalone hardware components on programmable logical circuits;
- I also support the network based communication of components via DDS, as well as other forms of data-driven communication such as shared memory and memory mapping commonly used by safety-critical, real-time embedded systems.

The rest of the work is structured as follows. Chapter 2 introduces the concepts that are necessary to understand the rest of the work. Chapter 3 presents the overview of the approach used in this work to support the model-driven development of heterogeneous cyber-physical systems. Chapter 4 introduces the existing code generation techniques of Gamma along with the newly developed code generators. Chapter 5 presents the new, data-centric semantic variant, its possible uses, and the new code generator developed to support distributed communication. Chapter 6 details notable implementation techniques and details used in the new code generators. Chapter 7 presents case studies that utilise my novel extensions of Gamma. Finally, Chapter 8 forms a conclusion, and presents possible future works.

Chapter 2

Background

This chapter presents the background information necessary to understand my work. It starts with the definition of cyber-physical systems (Section 2.1) and the introduction of model-driven development (Section 2.2). Then, it presents the Gamma Statechart Composition Framework (Section 2.3), which is a modeling tool this work builds upon. Next, the chapter describes the Data Distribution Service (DDS) standard (Section 2.4), a communication standard commonly implemented in distributed and heterogeneous systems. Finally, Section 2.5 presents some modeling tools and approaches related to this work.

2.1 Cyber-Physical Systems

The term cyber-physical systems (CPS) [21] refers to a new generation of systems which integrates computation with physical capabilities. These systems have the ability to interact with, and expand the capabilities of the physical world through computation, communication and control [22], while also being strongly connected to human users. CPS are usually feedback systems that have algorithms relying on sensor data to issue commands to the actuators controlling the physical parts of the system. Essentially, these systems have to integrate the dynamic behaviour of physical processes with the computational abilities of software and networking. Another common property of cyber-physical systems is heterogeneity. These systems can contain a wide-range of components, from embedded systems, which fulfill task-specific computation with limited resources, to cloud-based solutions with high performance and power. Components are often networked and distributed, and have to interact with each other in order to carry out tasks, some of which requiring real-time, intelligent and autonomous behaviour. As cyber-physical systems focus on the intersection of the cyber and the physical world and can interfere with human activity, they are often designed to be reliable, fault-tolerant and safety-critical. A comprehensive model of the properties of CPS is presented in Figure 2.1 [1]. The many parts found in such a system also have to be secure and resistant to malicious attacks, as even a single exploit can lead to catastrophic results. Examples of these systems can be found in medical fields, autonomous vehicles, Industrial Internet of Things and military.

Due to the heterogeneity of its components and the performance and safety requirements, the development of a cyber-physical system is a challenging task. Each component has to be designed precisely to comply with the requirements and the composition and networking of these components also have to be addressed, which brings a whole new array of possible errors and failures. To cope with the complexity, software developers and system engineers often rely on model-driven development approaches.

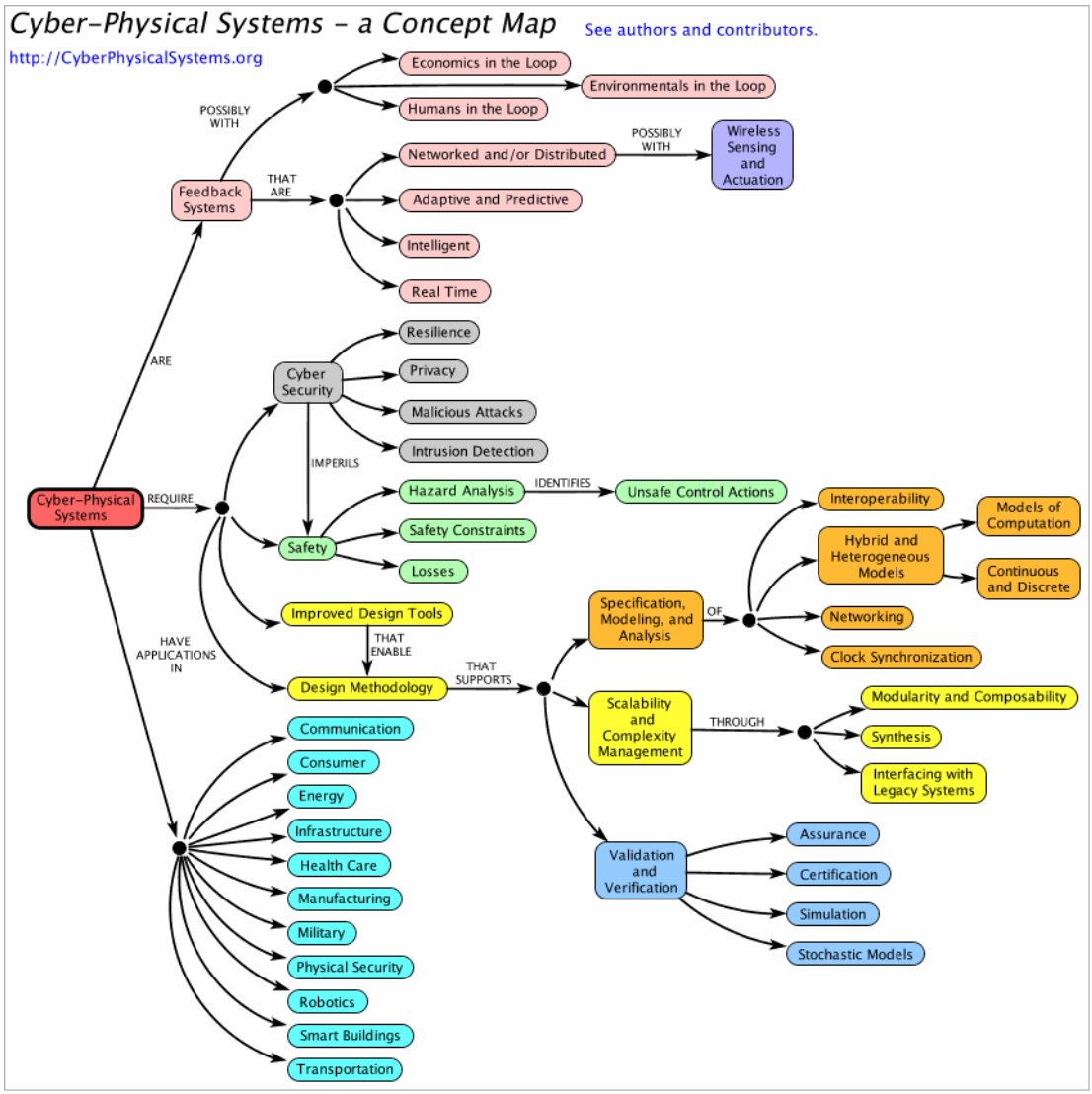

Figure 2.1: The concept-map of cyber-physical systems

2.2 Model-Driven Development

The basis of model-driven development (MDD) (or model-driven software engineering [5]) is the application of models to design various aspects of the systems [24]. The major advantage of model-driven approaches is that developers can focus on the high-level, logical problems and behaviour of the system, making the development less dependent on the platforms and technologies used. Models are also easier to understand and to maintain than text-based descriptions, and can be used to derive documentation or source code, which can reduce development costs and time [24]. This approach is often used in safety-critical system development [9], as models can precisely describe system behaviour and structure, while the derivation of source code and the early appliance of verification and validation (V&V) techniques minimize human error. This work builds on the platform-based design approach, a subset of model-driven development.

2.2.1 Platform-Based Design

The platform-based design approach [19] is a model-driven approach where users define the functional and behavioural models of the system in a platform-independent way, then connect and allocate these models to the hardware. This way, the platforms where the models are implemented appear in the design process. This approach is especially useful when designing complex embedded and cyber-physical systems, where there are multiple different platforms used. This way, developers can minimize the amount of human error by reducing complexity, while precisely defining the functionalities and interactions. A comprehensive visualisation of the models used in platform-based design is shown in Figure 2.2. The bases of these models are the requirements that the system needs to fulfill. These requirements usually start from a high-level, customer-based perspective, and can be refined to a low-level version, where the source code can be implemented from them without further information. From the high-level requirements, the functional model can be derived. Since the creation of new hardware is expensive, companies usually rely on generic hardware platforms when designing safety-critical systems. Therefore, platform models are dependent on their respective hardware library as well.

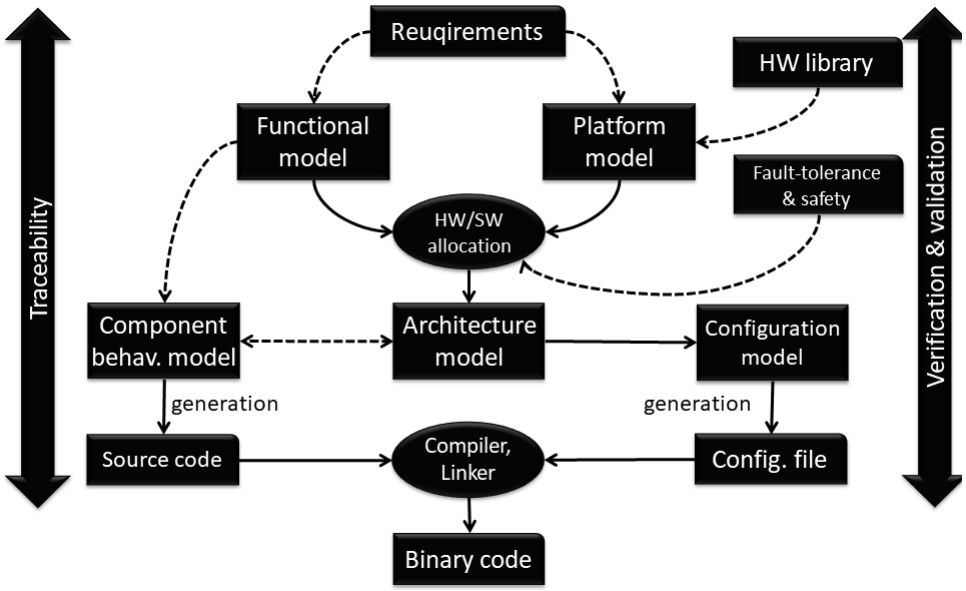

Figure 2.2: Platform-based system design

Functional models describe the functions and services inside a system, along with their interfaces and connections. Functional models also show the components of a system in addition the the inputs and outputs they have in accordance with the requirements. Platform models describe the structure and functions of the platforms that the system is built on. Component behaviour models define the logical behaviours of each component. They can model the internal behaviour of a component, for example, with statecharts, or the communication processes of these components with sequence diagrams. A sufficiently defined behaviour model can be used do derive implementation. From these, the hardware/software allocation and architecture models can be constructed, which describe what platforms the components run on as well as how they communicate with each other.

However, these steps vary greatly, and as such, developers usually have to resort to using multiple tools at the different steps of development. There is a lack of a single tool that supports all the steps depicted in Figure 2.2. In this work, our main focus is to support the development steps on the left side of the aforementioned model, where the description of

the behaviour and communication of the components happens. These are the steps that the Gamma Statechart Composition Framework supports with its functional and component behavioural modeling, while also providing traceability between model transformations and verification. Architectural modeling, which could aid the allocation and deployment of components to different platforms is only slightly supported by Gamma, but is planned to be developed as future work.

2.3 Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework [20] is an Eclipse-based, integrated modeling and analysis toolset for the component-based design of reactive systems. Gamma supports the semantically well-founded composition and analysis of heterogeneous statechart [15] components where individual components may use different statechart semantics. 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 on the basis of precise execution and interaction semantics [11] including scheduling strategies and constraints. 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 (V&V) 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.

This work focuses on the model-driven development of heterogeneous CPS with Gamma, using the composite component-based modeling approach provided by the framework to model such systems. Essential to the precise modeling of composite systems is to define the way components interact with each other, especially when heterogeneity is present. Gamma presents synchronous and asynchronous system compositions to describe component behaviours.

2.3.1 Composition Modes of Gamma

Gamma supports synchronous and asynchronous systems to define how composite systems behave, and how components inside such systems interact with each other. This subsection details these two types of systems along with their supported composition modes as well as some limitations I aim to address in this work.

Synchronous systems Synchronous systems operate with logical time, following the semantics of synchronous programming languages. The components in these systems communicate by sending signals through ports, and their execution is controlled by a clock that emits ticks. The execution of all components within a system is called a cycle. When executed, components first sample their input ports, then provide output on their output ports. Input signals are only sampled in the beginning, that is, if a signal changes mid-execution, the change is ignored, and will only take effect in the next cycle. Output signals are sustained until the next cycle. A unique aspect of synchronous systems is that components can react to the absence of inputs as well. Synchronous systems can be composed using the synchronous-reactive and cascade composition modes.

The *synchronous-reactive* composition mode describes components that are executed concurrently, in a lock-step fashion upon every tick. These components sample their input in the beginning of the cycle and process them at the same time. For this reason, communication between components during a cycle is not possible, changes in signals will only take effect in the next execution cycle. The *cascade* composition mode supports the sequential execution of components, i.e. they are are executed in a given order (the default being the order of declaration). Components read and process inputs not at the beginning of the cycle, but just before they are executed. This allows for the communication of the components during an execution cycle. Additionally, multiple execution of the same component in a single cycle is also supported.

Asynchronous systems Asynchronous systems represent components that operate concurrently and communicate through message queues. Writing to a message queue succeeds instantly, while reading from an empty queue blocks the reader. Messages arrive in the queue in the order they were sent, and the delivery of these is considered reliable, that is the sender does not receive confirmation. Reading always retrieves a single message from the queue. Queues can also have priority levels, (a higher number representing a higher priority) creating an order of reading. When a full queue receives a message, the message gets discarded. Currently, only one composition mode is supported, which is *asynchronous-reactive*, where components are running continuously, while also reacting independently to incoming events. It builds on the asynchronous adapter component type that wraps synchronous components and adapts them to the behavior of asynchronous systems.

Limitations of the asynchronous semantics Asynchronous behavior is common in heterogeneous cyber-physical systems, as sensor readings, data processing and communication between components happens at a different rate, while components run independently from each other. However, in most CPS, communication mostly happens using shared data, i.e. data is considered to be the main form of communication between components. This usually happens through some sort of shared memory space, where components write and read the same memory, seeing the same entries and their changes. Currently, the asynchronous semantics in Gamma do not fully support this behaviour, and as a consequence, the development of cyber-physical systems. In this work, I address this issue by introducing a data-centric semantic variant for asynchronous systems on the basis of the data-centric component type, a variant for the asynchronous adapter component type.

2.3.2 Formal Composition Semantics of Gamma

At its core, Gamma provides the Gamma Composition Language, which supports the formal composition of components according to different execution and interaction semantics (the synchronous-reactive and cascade composition modes in synchronous systems, and asynchronous-reactive composition mode in asynchronous systems). In this work, I propose and formalize a new semantic variant for asynchronous systems, which builds on the current formalization. Therefore, in this section I briefly overview the formal structures of the current formalization that are necessary to interpret the new semantic variant, that is, *events*, *event vectors*, *synchronous components*, *event sequences* and *asynchronous components*. For a more comprehensive description of the semantics, the reader is directed to [11].

Events The event definition below models a specific event of a specific port on a specific component instance.

Definition 1. An *event* is an observable phenomenon that can occur, such as the reception of a message, the change of situation (state) or the value assignment of a variable. Given a set of events E, the finite domains of event parameters are defined by the domain function $\mathcal{D} : E \to 2^{\{d_1,\ldots,d_n\}}$. The domain of an event $e \in E$ is $\mathcal{D}(e)$, a set of possible parameter values for event e. We say that an event $e \in E$ is *parameterized* if $|\mathcal{D}(e)| > 1$. An *instance* of an event is (e, p), i.e. the event with a specific parameter value $p \in \mathcal{D}(e)$. The set of all event instances for a given event e is denoted by $inst(e) = \{(e, p) \mid p \in \mathcal{D}(e)\}$. In case the absence of an event is of interest, $inst_\perp(e)$ is defined as $inst(e) \cup \{(e, \perp)\}$, where $(e, \perp)$ is the "null" instance that denotes the absence of the event. Finally, the set of event instances for events in a set E is $inst(E) = \bigcup_{e \in E} inst(e)$ (and $inst_\perp(E)$ similarly). ▪

Event Vectors In the synchronous domain, components communicate via signals. The formal structure describing signals is the event vector. An event vector can be regarded as a set of cells that can be filled with event instances, at most one instance in every cell. Event vectors are the inputs and outputs of synchronous components.

Definition 2. Given a set of events E, an *event vector* v_E is a function that assigns a (possibly "null") event instance to every event $e \in E$ such that $v_E(e) \in inst_\perp(e)$. The set of all possible event vectors is denoted by V_E. ▪

Synchronous Component The following definition specifies the formal syntactic contract of synchronous components. A synchronous component should have a set of states, a well-defined initial state, a set of input and output events (collected from ports of the component) along with their parameter domains (i.e. data type), and a deterministic transition function that describes the behavior of the component, which can be specified arbitrarily.

Definition 3. A synchronous component is a tuple $\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- S is the set of potential states, with $s^0 \in S$ being the initial state.
- I is the set of input events and O is the set of output events such that $I \cap O = \emptyset$. The set of all events is denoted by $E = I \cup O$.
- $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$ is the domain function of the events.
- $T \colon S \times V_I \to S \times V_O$ is the transition function, which determines the next state and the output event vector of the component when executing it in a given state with a given input event vector. Note that this definition requires the component to have a deterministic behavior.[1] ▪

Event Sequences In asynchronous systems, event vectors are substituted by event sequences.

Definition 4. An *event sequence* $q = \langle (e_1, p_1), \ldots, (e_n, p_n) \rangle$ is a finite, possibly empty (denoted by ε) sequence of event instances. The set of all possible event sequences for a set of events E is denoted by $inst(E)^*$, while $|q|$ denotes the length of the sequence. The ith event instance in the sequence is denoted by $q[i] = (e_i, p_i)$. Finally, a permutation of a set of event instances A is a sequence denoted by $\sigma(A)$ and all possible permutations of A is denoted by $S_\sigma(A)$. ▪

[1] Again, the definitions could be extended to nondeterministic models.

Asynchronous Component Asynchronous components are syntactically very similar to synchronous components. The only difference is the definition of transitions: it is now not a function but a relation, and instead of taking and producing an event vector, it takes a single event instance and produces an event sequence chosen from the potential output sequences nondeterministically.

Definition 5. An asynchronous component is a tuple $\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- S is the set of potential states, with $s^0 \in S$ being the initial state.
- I is the set of input events and O is the set of output events such that $I \cap O = \emptyset$. The set of all events is denoted by $E = I \cup O$.
- $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$ is the domain of the events.
- $T \subseteq S \times inst(I) \times S \times inst(O)^*$ is the transition relation, which determines the possible next states and the possible sequences of output events of the component ($inst(O)^*$) when executing it in a given state with a given input event. Note that this definition *does not* require deterministic behavior. ▪

2.4 Data Distribution Service

The Data Distribution Service (DDS) [12] is a communication middleware standard developed by the Object Management Group (OMG) that is widely used in critical and Internet of Things systems to distribute and share data. It utilises a global data space, creating a shared-memory like distributed database among participants. It is based on a publish-subscribe model, where components can publish data to subscribers without actually knowing who will receive the data. DDS provides dependable, high-performance, scaleable data exchanges between real-time components. There are multiple open-source and commercial DDS implementations and vendors, which can differ in target platforms and Quality of Service metrics.

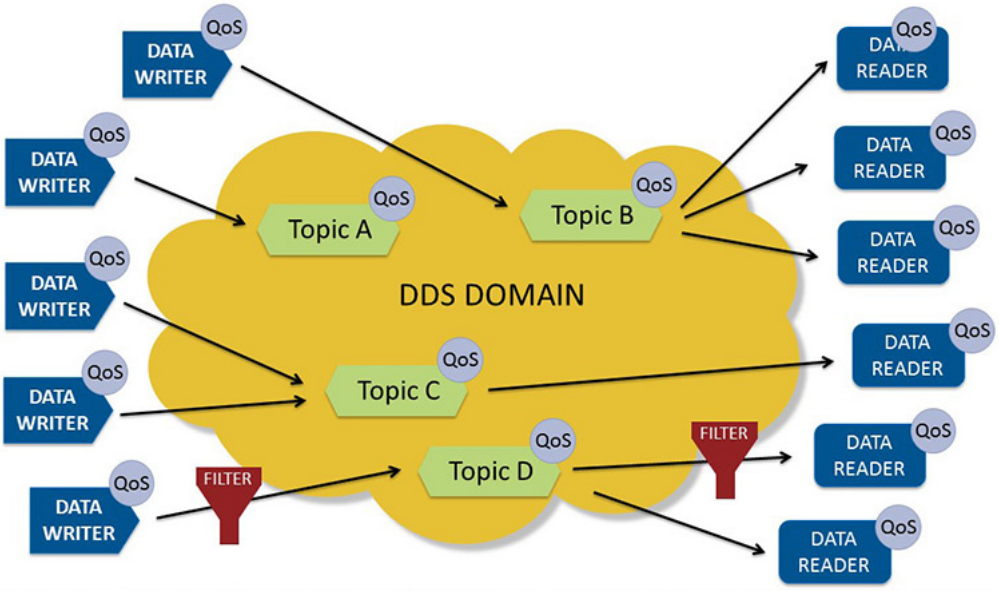

Figure 2.3: The core functional concept of DDS [2]

A typical structure of a DDS-based communication network is depicted in Figure 2.3. The outermost layer is the *domain*. Domains define which applications can see each other. An entity that is present inside a domain is called a *participant*. Participants can only interact with each other if they are within the same domain, as domains are independent from each other (although applications can have multiple participants that can be in different domains, allowing communication in more than one domain). Applications send data to and receive data from specific *topics*, which are sets of global variables with the same types. Applications can publish and subscribe to different topics, where *publishers* send data to all *subscribers*. Publishers write data to topics via *data writers*, which create samples of a single data type. Subscribers use *data readers* to access data, which receive samples of a single data type. Both publishers and subscribers can have multiple data writers or data readers, writing to and reading from different topics. Topics can also have many data readers and data writers assigned to them, so multiple applications can read or write to the same topic. Data readers and writers can also filter topics, specifying data that is interesting to them. So as an example, a temperature sensor application can send its data in a specific domain using a publisher (which is considered a participant) that can have one or more data writers that write the data to different topics.

Most elements found in a DDS domain can have Quality of Service (QoS) properties assigned to them. QoS properties are a set of configurable parameters that control the behaviour of the DDS system, such as reliability, resource consumption and fault-tolerance. Domain participants, topics, publishers, subscribers, data writers and data readers all have QoS policies. There are DDS entity specific policies, and also general ones, which have the same effect on all entities. For example, the History QoS manages how sent/received data is stored within a data writer or data reader, taking the applied resource limits into consideration (which is another QoS property). Only entities with compatible QoS properties can cooperate (otherwise interaction between them is impossible). For example, a data writer with a BEST_EFFORT *Reliability* QoS cannot send data to a data reader with RELIABLE *Reliability*.

2.4.1 RTI Connext DDS

In this work, RTI Connext DDS [18] is used, which is a popular choice when it comes to Industrial Internet of Things (IIoT), medical devices and autonomous vehicles. RTI Connext DDS provides a wide range of QoS parameters and distributes data on the DDS Databus [23], as special data distribution implementation by RTI. With the Databus, applications can communicate by simply reading and writing data object within the global data space (the DDS *domain*). The DDS middleware maintains the data space, which captures a system's state. Essentially, applications do not message to each other, instead, they write data to the share data objects found in the bus. This data-centric publish/-subscribe approach also has an in-memory data management layer. Applications operate with their local caches, and DDS synchronises these caches by publishing updates and subscribing to data of interest.

RTI DDS also features code generators that can provide the necessary implementations from an Interface Description Language (IDL) [14] file, which defines the data structures and types that can be sent or received.

[2]https://www.dds-foundation.org/what-is-dds-3/

2.5 Related Work

ThingML In [8], a tool supporting a Model-Driven Software Engineering approach is presented, focusing on the heterogeneity and distribution challenges of cyber-physical systems. The ThingML language uses composite state machines to model the behaviour of components. It supports asynchronous communication (by sending messages through ports) and event-based reactive programming. The tool uses an action language to describe the actions within event processing rules and state machines. The paper demonstrates the wide range of platforms and areas of cyber-physical systems, while also showing the most popular languages used in each area. It supports the generation of implementation and communication, based on the aforementioned platforms and languages. However, ThingML does not support the automatic generation of hardware description languages, and also lacks the verification capabilities of Gamma. In addition, ThingML does not aim to support the development of critical systems.

xMAS In [6] a set of microarchitectural primitives is defined, which allows for the description of complete systems by composition alone. It focuses on the efficient system-level connection of different IP blocks, which is an integral part of SoC design. However, if not designed carefully, the interconnection of these IP blocks can lead to live- and deadlocks, which can be hard to debug, especially in a distributed system. The article proposes an approach based on creating executable and analyzable high-level models, called xMAS (eXecutable Microarchitectural Specification) models. These models utilise an extended set of microarchitectural primitives to describe complete systems, without using intermediate glue logic. For verification purposes, it generates Verilog out of the models and uses inhouse and academic tools for formal verification. This approach focuses on defining models on a microarchitectural level, rather than defining a high-level behaviour models, like Gamma.

In [28], a novel prototyping technique for concurrent control systems in FPGAs is presented. The method focuses on the dynamic reconfiguration of the system, which means that the functionality of part of the controller can be changed, while the rest of the system is running. The approach uses UML statecharts [13] to model the system, and it has been experimentally verified using a Xilinx FPGA. The UML statecharts are converted into concurrent finite state machines, basically flattening the statechart, removing the hierarchy elements. To describe the controller in an HDL, Verilog code is generated. This work demonstrates its approach on a home area network (HAN) controller, which is in charge of controlling smart home devices. This article shows how the model-driven approach is used when targeting embedded systems, and the capabilities and flexibility of FPGAs. This serves as an example for the possible usages of this work, showing that there is a need for modeling tools capable of designing complex embedded systems.

Chapter 3

Overview of the Approach

This chapter overviews my proposed approach for the design of heterogeneous CPS. My approach relies on the platform-based design and modeling approach, detailed in Section 2.2. The chapter demonstrates the general difficulties when using this design approach in heterogeneous cyber-physical systems, while also presenting the capabilities of the Gamma framework when targeting these platforms (Section 3.1). Moreover, this chapter presents my approach that introduces additional functionalities in Gamma, which allow the framework to better support the model-driven development of these systems (Section 3.2).

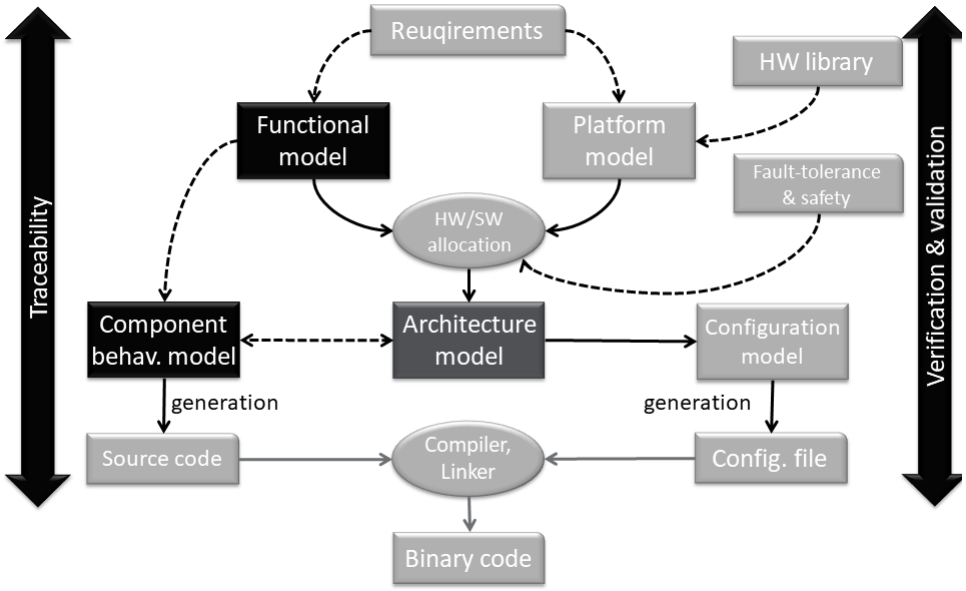

Figure 3.1: Platform-based design steps supported by Gamma

3.1 Difficulties of Developing CPS

Heterogeneous CPS are built from a wide variety of hardware and software components. System engineers need high-level languages to capture the behaviour and various functions of such systems. In addition, the communication has to be established reliably, so modeling and operating the communication between the distributed components need support. Engineers might need to resort to a wide variety of tools to comply with the different needs of the multiple used platforms. As an example, the Zynq-7000 System-on-Chip integrated

on the Digilent ZedBoard consists of dual-core ARM processors and a Xilinx FPGA (visualised in Figure 3.2). This board can connect to a network via Ethernet, receiving and sending data from/to a different computer. The platform-based modeling of a system where the components run on different parts of a heterogeneous system is challenging, as the programming languages and communication modes can be entirely different, yet these parts have to interact with each other.

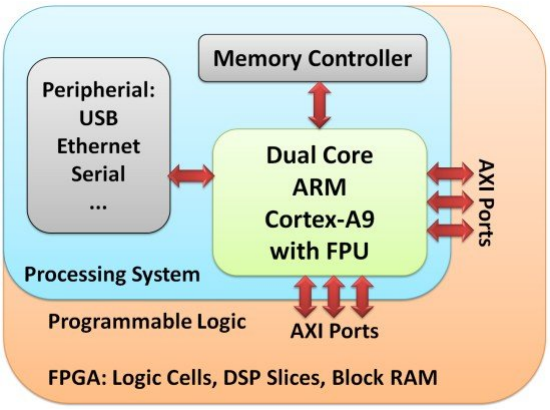

Figure 3.2: A high-level model [16] of the Zynq-7000 SoC

The Gamma Statechart Composition Framework supports the platform-independent modeling of components in a homogeneous environment (its supported modeling steps are presented in Figure 3.1). However, currently Gamma does not fully support the development of CPS, as it lacks the support of multiple platforms and architectures. The main goal of this work is to extend the capabilities of the Gamma Statechart Composition Framework to support the development of such systems, by

- being able to model composite CPS,
- supporting various architectures used in CPS,
- being able to establish communication between components running on different targets.

3.2 Design and Code Generation for Heterogeneous CPS

My first goal is to extend the Gamma framework to support automatic implementation of heterogeneous CPS. In this section I overview the modeling and automatic code generation techniques for embedded microcomputers, FPGAs and networks of these components. My choice of target systems was motivated by the fact that these languages are widely used in critical cyber-physical systems.

The modeling steps of the platform-based design approach (depicted in Figure 2.2) supported by this work are shown in Figure 3.3. First, code generators are developed that support the definition of component behaviour in multiple languages, creating the opportunity to support multiple architectures and hardware platforms. Considering the functional model, a new communication method is introduced with a corresponding semantics to correctly describe component interactions within the composite system. The hardware/software allocation and architecture model steps are only indirectly affected, as the extensions introduced in this work affect the capabilities of Gamma when it comes to supporting new platforms.

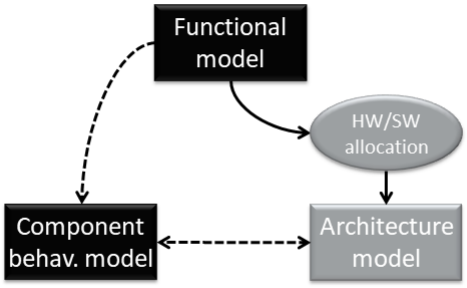

Figure 3.3: The modeling steps improved in my work

3.2.1 Modeling of Composite CPS

One of the most crucial parts of designing a composite system is to precisely define the interaction of components. This includes not only the connections but also the manner and timing of the communication. These attributes of the system can be linked to its functional model. When it comes to the description of the functional model, Gamma uses its composition language, named Gamma Composition Language (GCL) to describe how components are connected and how they communicate. This is achieved by supporting multiple composition semantics and composition modes that allow for the description of synchronous, cascade, and asynchronous composite components.

In synchronous systems the flow of information is controlled. On the other side, asynchronous composition lets the components run independently with little guaranties regarding the reception and processing of data. In critical CPS, we have to rely on the communication, which can be achieved by using reliable communication middleware such as DDS. DDS implements a shared-memory like communication with various assurance mechanisms. This type of communication is a common type of cooperation in critical CPS (especially running on a heterogeneous platform), where components utilise a shared-memory to communicate an share data. This form of composition needs modeling support. My approach presents this *data-centric semantic variant for asynchronous systems*, where:

- components interact with each other through shared variables,
- shared variables have well defined writers and readers,
- components run independently of each other.

With the introduction of this variant, the behaviour of components inside a heterogeneous CPS can be described accurately. Using DDS, it is possible to realise a composition where components can run independently, but can communicate with customisable reliability (utilising the QoS options of DDS) through a shared-memory.

3.2.2 Supporting New Architectures and Platforms

Modeling tools use component description languages to define the behaviour and interfaces of components. In Gamma, the Gamma Statechart Language (GSL) is used to define atomic components using statecharts. This model description can be used to derive implementation, currently in Java. When it comes to cyber-physical systems, Java is commonly

used in non safety-critical components, such as cloud. However, in embedded systems, Java is rarely used, mainly because of its memory requirements.

As a typical CPS consists of many embedded platforms, it is necessary to support the development on top of these devices. To solve this, I introduce new code generators, which allow for the derivation of source code from the component statechart definitions in C and SystemVerilog languages. C is one of the most prevalent languages when it comes to embedded devices, and is widely supported. SystemVerilog is a hardware description language, supporting the development on FPGAs. Both of these languages are commonly used when developing on embedded platforms, allowing Gamma to generate implementation to a wider variety of architectures. Currently, these code generators support the derivation of source code from synchronous and cascade composite components.

3.2.3 Establishing Communication

With the introduction of vastly different platforms and devices, the problem of reliable and easy communication emerges. CPS often rely on communication middlewares to solve this issue. Middlewares in distributed systems are software layers in between the operating system and the applications. They essentially allow developers to focus on the actual purpose of applications, as middlewares handle the passing of data and communication between different components. A commonly used standard for this purpose is the Data Distribution Service (DDS). In this work, RTI Connext DDS is used to extend the communication capabilities of Gamma, which allows the usage of a shared-memory to convey data between components in a reliable manner.

The extensions introduced in this work (comprehensively shown in Figure 3.4) allow the Gamma framework to

- target a wider variety of platforms,
- establish networked communication between components running on different targets,
- precisely define the interactions between these components when data-centric communication is used.

These new features facilitate the model-driven development of heterogeneous cyber-physical systems using Gamma.

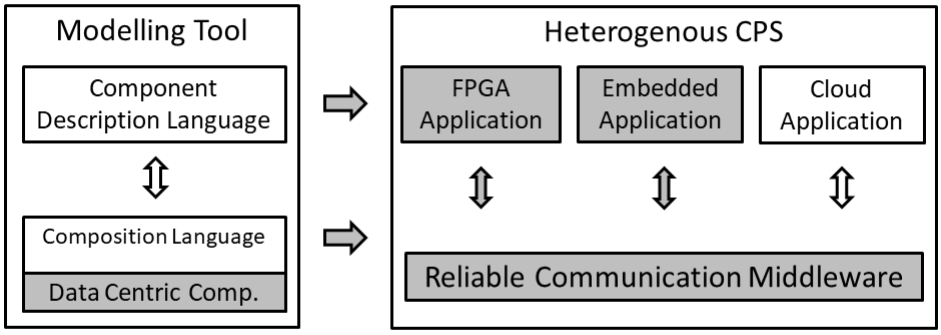

Figure 3.4: CPS development areas targeted by this work

Chapter 4

Modeling and Code Generation for Standalone Hardware

This chapter presents the modeling and code generation processes with the introduced generators when targeting standalone hardware. First, it briefly describes the elements of Gamma statecharts (Section 4.1). Second, it presents the way Gamma transforms high-level models to low-level ones (Section 4.2). Finally, the chapter presents how these low-level component descriptions are used to derive implementation both in C (Section 4.3.1) and SystemVerilog (Section 4.3.2), and how these implementations function.

4.1 Gamma Statecharts

In Gamma, components are defined using statecharts (an example is presented in Figure 4.1), which are based on the Unified Modeling Language (UML) [13] standard. Statecharts contain states, which define the stable situations of the component as well as how it reacts to incoming events. Events are happenings in the component that states can process and react to, possibly resulting in state changes in the component. Events can have parameters, which contain additional information, and can have altered values in different event instances. Events can come from outside the component, or can be produced by internal timers. States and transitions can have timeout events associated to them, which fire after the passing of a set amount of time while the component is in the given state. States can also have entry and exit actions, which are executed upon entering or leaving a state, and can possibly fire events. Hierarchy is supported as well, that is, states can have other states embedded inside them, refining state behaviour. Orthogonal regions are also present in Gamma statechart, allowing the definition of concurrent behaviour. Gamma also supports shallow and deep history states, which store the last active state contained in a higher-level state upon leaving, i.e. upon reentering the high level-state, the active inner state will be the one stored by the history state.

Statecharts are high-level behavioural models, as they define the logic of the system, and the way it executes the necessary steps in order to function. Even though, manual implementation of logical behaviour is possible, it is a tedious task, which can be mitigated by automatic code generation. However, when considering automatic source code generation, the problem of transformation between languages emerges. Code generators need to work on the high-level model, but the high-level elements introduced in statecharts to make the model more readable to humans can pose a challenge during the generation

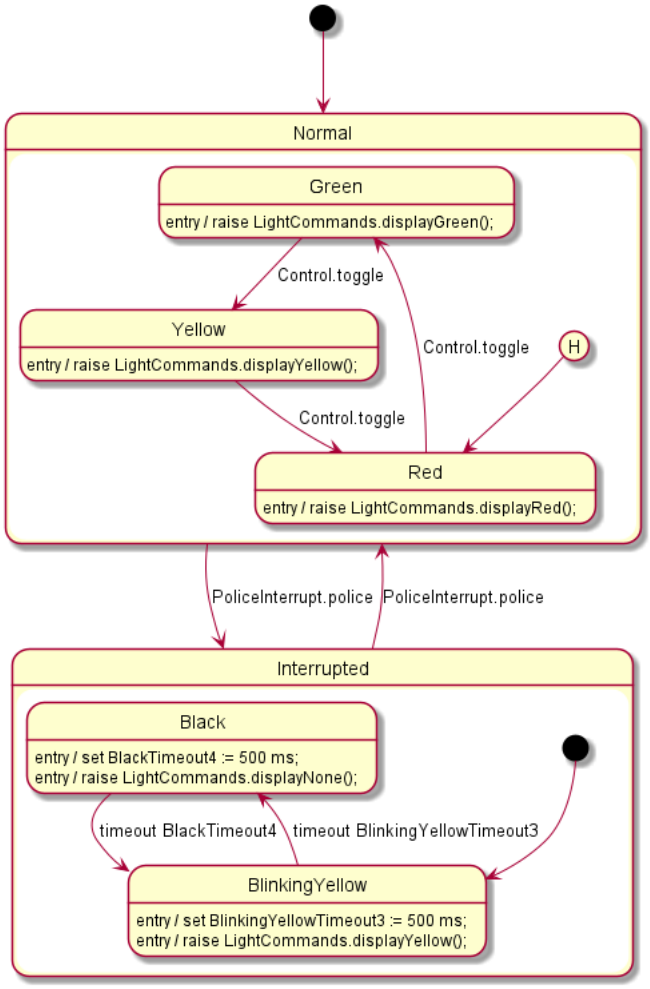

Figure 4.1: A traffic light with police interruption, modeled as a Gamma statechart

process. Gamma solves this issue by transforming its high-level models to low-level model descriptions, bridging the abstraction gap that would make code generation difficult.

4.2 High-Level Model Transformation

The model transformation functionalities of Gamma is presented in Figure 4.2. Gamma uses two lower-level modeling languages, each with a different abstraction level, to facilitate the transformation from a high-level model. The first, the Low-Level Statechart Modeling Language which the high-level statechart gets transformed to, essentially serves as an intermediate step. This work does not utilise the Low-Level Statechart Modeling Language, so further information about it is not provided. This Low-Level Statechart Modeling Language then gets transformed into an Extended Symbolic Transition System (XSTS) modeling language, which is then used to create code generators.

The XSTS modeling language describes a statechart with transitions and variables, focusing on the behaviour of the model without the high-level parts. It essentially flattens out hierarchies from statechart. Events are transformed into variables, that are modified based upon the handling of their corresponding event. Timeout events are represented with numerical variables, that get incremented as the time passes (based on clock ticks),

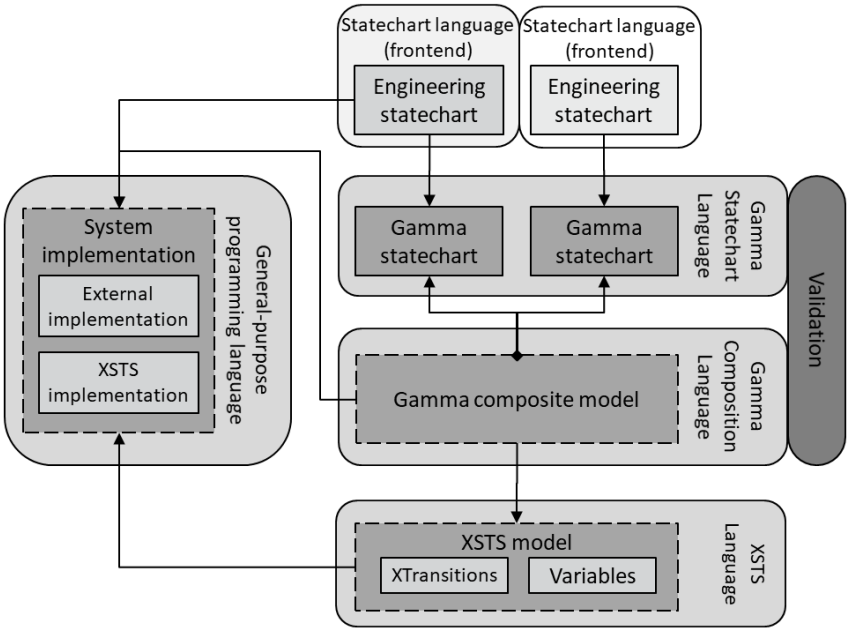

Figure 4.2: An overview of the model transformation and code generation functionalities in Gamma

while non-timeout events are mapped into boolean type variables. States are represented by enumeration variables. For every region, an enum variable is created, with its literals being the states contained inside (which can also be composite, having their own enum variable). There is also an enum variable for the top layer of statechart, where the literals are the topmost states of the model. An *_Inactive_* literal is also generated in every enum type, indicating that the corresponding region is not active.

The code generators introduced in this work take an XSTS model as input and exploit the low-level structures of the XSTS language to generate the core behaviors (the execution of a single step) of components. For the description of these low-level structures, only *conditions* (*if-else* structures) and variable assignments have to be supported, facilitating the integration of almost any programming languages. Thus, in different languages and platforms, developers can focus on the infrastructure (such as input and output event handling and timings according to the semantics) of the generated implementation.

A model described with XSTS can be processed in two different ways, resulting in two code generation techniques. These techniques optimize different parts of the XSTS model. The *VariableCommonizer* keeps the structure of decision branches, resulting in a smaller source code. The other, the *ChoinceInliner* implements a switch-case structure, which is faster if there are a lot of branches that can be optimized. However, the resulting source code is substantially larger compared to the *VariableCommonizer* when used on large models. Essentially, the two techniques create different implementations. When translating an XSTS model to the desired language, these differences usually appear in the functions that control the statechart. The different implementations, however, will produce the same results when considering the behaviour of the model.

4.3 Generating Implementation

Both code generators introduced in this work currently support the automatic code generation from synchronous-reactive and cascade composition modes. It is possible to generate implementation from a single statechart, or a composition of statecharts (an overview of the process is illustrated in Figure 4.3 for C, and in Figure 4.5 for SystemVerilog). Gamma supports the automatic transformation of synchronous-reactive and cascade composite components into XSTS models when generating source code, that is, the resulting implementation is a combination of all statecharts found within the composition. Therefore, the implementation generated from synchronous and cascade composite components is similar to that of single statecharts. The key difference is that composite component implementations contain all the statecharts within them, i.e. all event variables are generated from all components, and the implementation executes operations on the whole composition, not just a single statechart (demonstrated in Figure 4.4 for C, and in Figure 4.6 for SystemVerilog). Considering the two supported composition modes, the differences between synchronous-reactive and cascade are handled in the XSTS model, so the introduced code generators do not have to take their different characteristics into account.

4.3.1 C Code Generation

C is one of the most popular and widely used programming languages when it comes to embedded systems [8]. Therefore, in order to support the development of cyber-physical systems, I chose to extend the Gamma framework with a C generator.

Currently, synchronous-reactive and cascade composition modes are supported when generating implementation in C. When generating code from a single statechart, two pairs of source code and header files are generated, which contain the functions that realise the functionalities of the model. At its core, the derived implementation contains a structure, that encompasses all the variables associated with the model: states, non-timeout and timeout events. Multiple functions are generated as well, which perform operations on the model, like switching states or resetting the implementation to its initial state. All these functions perform operations on the structure that contains the model variables. Generating implementation for composite components produces files the same way as generating for standalone components does, and functions identically to them. The difference is that structures contain variables from all the component statecharts, and functions execute operations on the whole system.

Since the generated implementation operates with functions to control the statecharts, the different implementations introduced in Section 4.2 can cause alterations in the way the implemented model is controlled. To hide these differences, a wrapper structure is introduced. This contains the statechart or composition structures, and variables that implement the timing mechanisms. Functions are also generated to execute operations on the wrapper structure, and as such, on the implementation as well.

The resulting implementation of the model can be utilised by creating an instance of this wrapper structure, and using the functions generated to execute steps on the statechart.

4.3.2 SystemVerilog Code Generation

SystemVerilog is a hardware description and verification language commonly used to model, design, simulate, test and implement electronic systems. Its predecessor, Ver-

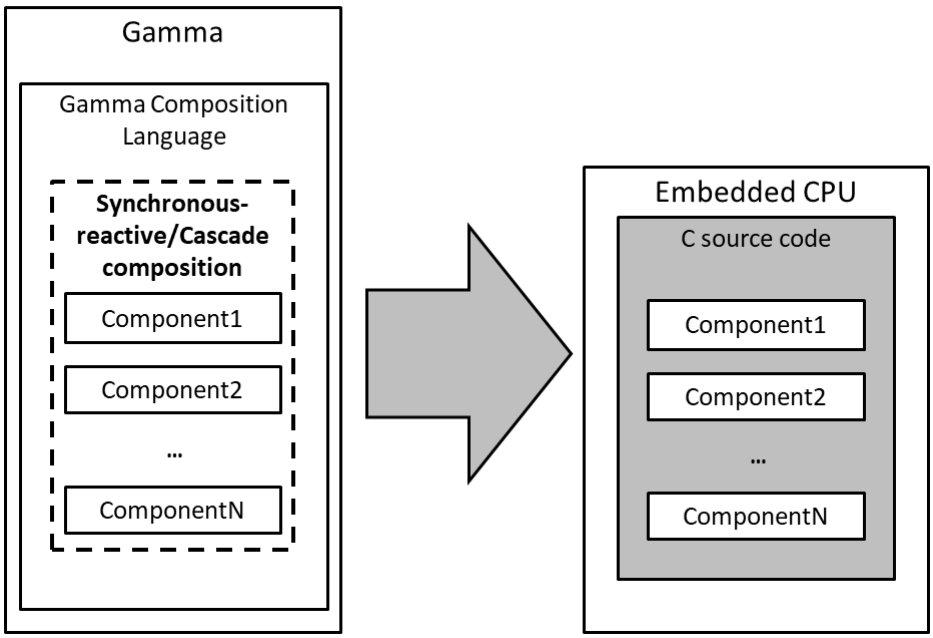

Figure 4.3: The process of generating C source code

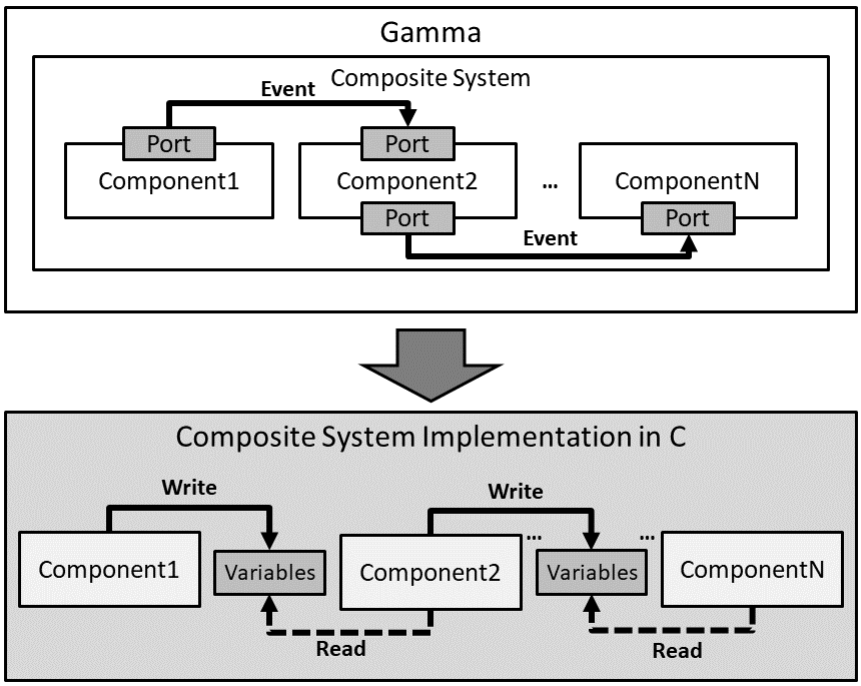

Figure 4.4: Communication between components in a composite component implementation in C

ilog, is one of the most popular languages when it comes to ASIC or FPGA development and prototyping [4]. It uses register-transfer level (RTL) design to describe the behaviour of registers and to perform logical operations on them. SystemVerilog extends these capabilities with elements resembling programming languages, like object-oriented design and

high-level language elements. These additions make the precise description of Gamma components easier.

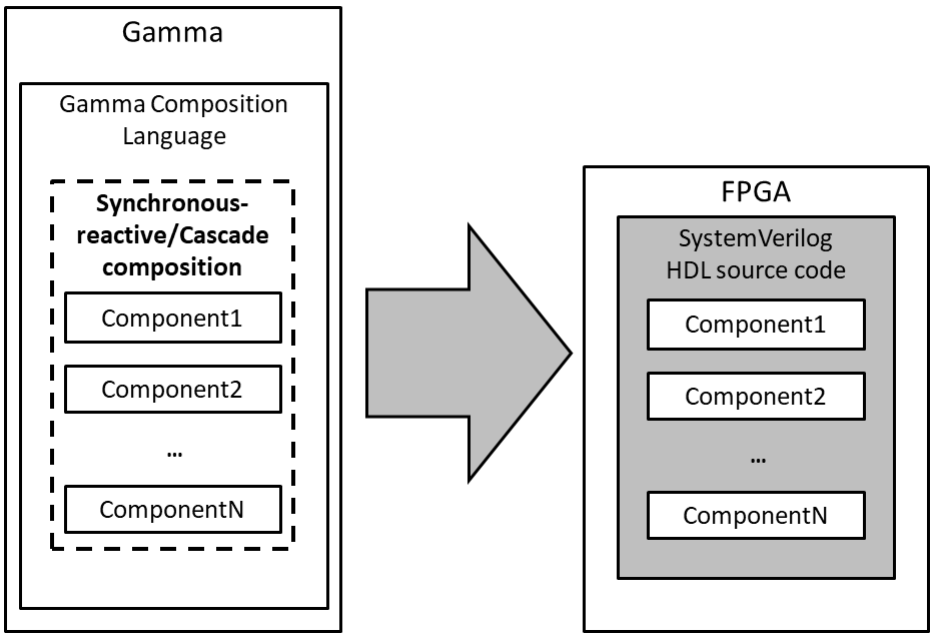

Figure 4.5: The process of generating FPGA HDL

In SystemVerilog, Gamma components are described as modules. Modules have input and output signals that can be connected to external sources such as GPIOs or other modules when instantiated; the input and output events of components are realised using such input and output signals. The input signals are processed inside the module, where the behaviour is described, and output signals are produced when the corresponding events are raised. Inside the module, the enumerations representing the states are generated as well. Unlike the C source code, the SystemVerilog implementation does not utilise functions to operate the model. Instead, it does every action in a clock-driven loop, changing the state of the model, and incrementing timeout variables.

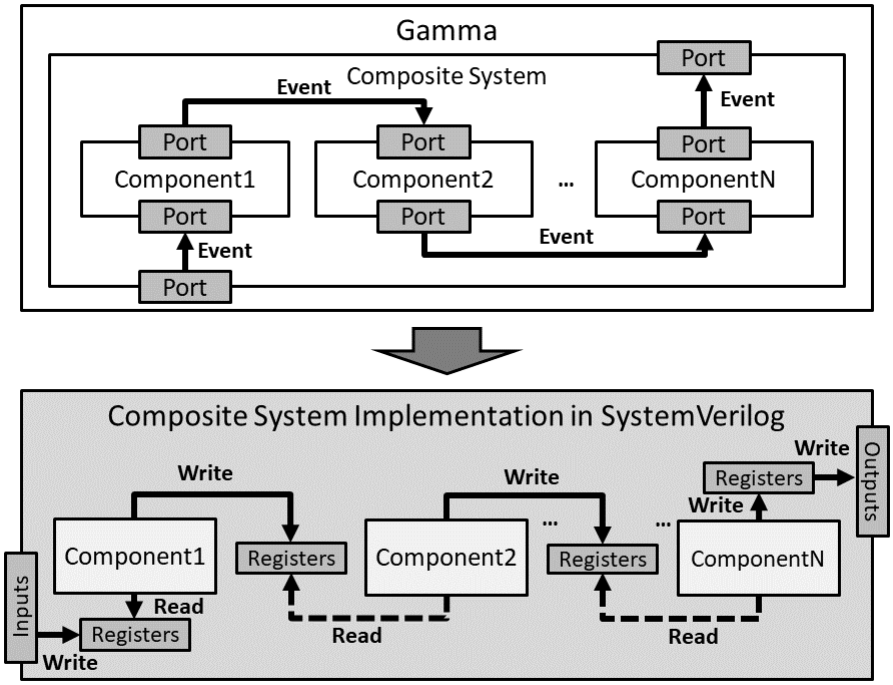

Figure 4.6: The realisation of a composite component in SystemVerilog

Chapter 5

Modeling and Code Generation for Distributed CPS

This chapter presents the modeling and code generation techniques introduced in this work when targeting distributed components in cyber-physical systems. First, it presents the data-centric semantic variant, the main theoretical novelty introduced in this work (Section 5.1). Second, the chapter gives examples of the possible implementation of the data-centric semantic variant (Section 5.2). Finally the chapter presents the data sharing process of DDS, and how this process can be used with the new semantics (Section 5.3).

5.1 Data-Centric Semantic Variant

To precisely describe communication within cyber-physical systems, I introduce a new semantic variant for asynchronous systems, the *data-centric semantic variant*. Note that these definition build on and complement the formal structures introduced in Section 2.3.2.

5.1.1 Formal Definition

The data-centric semantic variant restricts events to having only one parameter, as data-centric components define value assignments to variables by sending events with the corresponding value as a parameter.

Definition 6. Given a set of events E, the finite domains of event parameters are defined by the domain function $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$. The domain of an event $e \in E$ is $\mathcal{D}(e)$, a set of possible parameter values for event e. ▪

A data-centric component wraps a single synchronous component and converts it into the asynchronous domain. It can be considered as the variant of the asynchronous adapter in which the *trigger predicate* is a trivial function: in data-centric components all incoming events serve as trigger specifications.

Definition 7. A data-centric component for a synchronous component is defined as a tuple $\textcircled{D} = (\ominus)$:

- $\ominus = (S_s, s_s^0, I_s, O_s, \mathcal{D}_s, T_s)$ is the wrapped synchronous component.

Semantics. The semantics of data-centric components is defined in terms of an asynchronous component. From an external point of view (observed from the environment), a data-centric component processes input events one-by-one (just like asynchronous components in general), but may not always produce an output. The role of the data-centric component is to "collect" messages, feed the collected messages to the wrapped synchronous component and emit messages created from the resulting output event vector.

Definition 8. A data-centric component Ⓓ for a synchronous component is itself an asynchronous component $Ⓓ\rangle\!\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- $S = S_s \times v_I$ is the set of potential states, each state consisting of a state of the wrapped synchronous component and a buffer input event vector collecting the incoming event instances.
- $s^0 = (s_s^0, \perp_I)$, where $\perp_I$ is the empty input vector.
- $I = I_s$ is the set of input events, i.e., the input events of the wrapped synchronous component. From an input vector v_I we can derive v_{Is} as $v_{Is}(e) = v_I(e)$ for every $e \in I_s$.
- $O = O_s$ is the set of output events defined in the wrapped synchronous component.
- $\mathcal{D} = \mathcal{D}_s$ is the domain function of the wrapped synchronous component.
- The transition relation is defined as a (nondeterministic) transition function $T((s_s, v_I), (e, p)) = \{(s'_s, v'_I)\} \times \Omega$, where e is a, such that:
  - The buffer input event vector is updated such that $v''_I(e) = (e, p)$ and $v''_I(e') = v_I(e')$ for every $e' \in I$ $(e \neq e')$, and s'_s should be such that $T_s(s_s, v''_I) = (s'_s, v_O)$, and $v'_I = \perp_I$. $\Omega = S_\sigma(\{(e, p) \mid v_O(e) = p, p \neq \perp\})$ (as the set of possible output sequences) is every possible permutation of the "non-null" elements of the output vector. ▪

Discussion. The data-centric semantic variant introduced in this work is a variant of the asynchronous semantics. Specifically, it proposes an altered behavior for asynchronous adapters. Data-centric components use data as a means of communication. Components share data using shared variables, each having one well-defined reader and zero or more writers. Shared variables are realised by events having only one parameter: the writing of a variable is realised by sending an event with a parameter value, whereas reading one is realised by reading the parameter of a received event. If there are multiple writes to the same variable (instances of the same event are sent multiple times to the same component), always the latest will prevail.

The data-centric semantic variant, as it extends asynchronous behavior, does not restrict the running frequency and runtime of components, that is, they can run independently from each other. The execution of components without reading or writing data is not supported. In order to make data-centric components fault tolerant, readers are notified when their corresponding writer goes absent from the communication. In case of losing a writer, a new one can be instantiated, if supported by the implementation.

5.2 Data-Centric Communication Implementations

This section gives examples of possible implementations of the newly introduced data-centric semantic variant. The type of communication where two applications or processes communicate with each other by sharing data is called Inter-Process Communication (IPC). Inter-process communication (IPC) can be found both in Linux [10] and in Windows [7] operating systems. Typically, applications can use IPC as clients or servers. A client is an application or process that requests data or service from the server, which is another application or process. Client and server roles are not strict, as an application can act as a client or a server, depending on the situation. The rest of the section presents different forms of IPC.

Named pipes: Pipes are forms of communication between one or more related or interrelated processes, e.g. between a child and a parent process [26]. Communication is achieved by one process writing to a pipe, and another reading from it. Pipes are unidirectional, meaning the reading process cannot write to the pipe. This approach does not work with unrelated processes, for example, between two processes running in different terminals. Named pipes (also known as FIFOs) solve this issue, by allowing bi-directional communication between unrelated processes.

File mapping: File mapping is the process of mapping files to the virtual memory address of an application [25]. This enables a process to treat the contents of a file as if they were a block of memory in the process's address space [7] . Processes can use simple pointer operations to examine and modify the contents of the file. There is also a special case of file mapping that allows the creation of a named shared memory, which is used between processes.

Shared memory: As implied by the name, a shared memory is basically a memory accessed by multiple processes at the same time [27]. Applications can work with the same data without redundancy, as they see and utilise the same memory. A similar approach is demonstrated in Section 7.2, where components communicate through shared memory spaces, which can be described with the newly introduced data-centric semantic variant.

DDS: Since DDS uses data as the means of communication between applications, it can be used to realise the data-centric semantic variant. DDS can function similarly to a shared memory (this is further elaborated in Section 5.3). My approach builds on RTI Connext DDS to implement the newly introduced data-centric semantic variant.

Although application in cloud is not discussed in this work, the data-centric semantic variant could be applied to cloud-based data exchanges as well.

5.3 Sharing Data With DDS

At its core, DDS operates with a global data space [12], which from the applications point of view, looks like a local memory accessed via an API. Applications read from and write to these local memories, but in reality, when it comes to writing data, DDS sends messages to update the memories of remote nodes. These local stores give the illusion of having access to a global data space to the applications. Essentially, the global data base is a virtual concept in DDS, and is realised as a collection of local data stores. Each node operates with its own local memory, storing only what it needs, as long as necessary.

This mode of communication can be considered a shared-memory communication, where applications have access to and operate with the same data entries.

In safety-critical CPS, reliable communication is essential. RTI DDS provides various QoS options that enhance the reliability of interactions. In Table 5.1, some of the more interesting QoS options are presented, along with their description and applicability in Gamma. Introducing DDS to Gamma allows the framework to implement asynchronous composite components in distributed networking, with QoS options aiding the enforcement of semantics. For example, the combination of *History* and *ResourceLimits* QoS options supports the DDS implementation of message queues used by asynchronous components.

The *Liveliness* QoS also allows users to design models with fault tolerance in mind, since this QoS option notifies data readers when one of their data writers ceases to communicate. This notification can be mapped to the raising of an event, and thus, it can processed by components at their discretion. As a result, QoS options not only allow Gamma to implement semantic rules, but give designers means to create fault-tolerant models that can react to events raised by DDS, facilitating the design of safety-critical systems.

QoS name	Description	Additional information	Applicability in Gamma
History	Controls how much data to store and how stored data is managed for a DataWriter or DataReader. Options: KEEP_ALL, KEEP_LAST (with corresponding depth value)	KEEP_ALL depends on the resource limitations defined in the ResourceLimits QoS.	Allows Gamma to implement message queues using DDS.
Liveliness	Configures the mechanism that allows DataReaders to detect when matching DataWriters become disconnected or dead.	The checking of liveliness can happen automatically or manually. Events generated: on_liveliness_lost for DataWritersListeners on_liveliness_changed for DataReaderListeners	Allows for fault-tolerant model design, e.g., the specification of error events.
Reliability	Enables reliability protocol for a DataWriter/DataReader connection.	Options are BEST_EFFORT and RELIABLE. It is usually used in conjunction with History and Reliability, to determine which data is remains relevant and restorable.	Enforces reliable communication between distributed components.
ResourceLimits	Controlling the amount of physical memory allocated for entities.	Usually used in conjunction with History and Reliability to determine the size limits.	Specifies the resource limits of message queues.

Table 5.1: RTI Connext DDS QoS and Gamma supportability. [17]

With some of the QoS options presented in Table 5.1, it is possible to implement the data-driven semantics using RTI Connext DDS.

- **History**: setting it to KEEP_LAST with depth of 1 ensures that only the most recent data entries are kept.
- **Reliability**: when set to RELIABLE, Connext DDS monitors sent data to make sure it is received, and resends data that was missed.
- **Liveliness**: with the DDS_AUTOMATIC_LIVELINESS_QOS setting, Connext DDS will automatically assert liveliness. This will notify DataReaders if their no longer alive.

5.4 Generating DDS Communication

Generating the implementation of DDS communication is currently supported for asynchronous composite components (depicted in Figure 5.1), in C language.

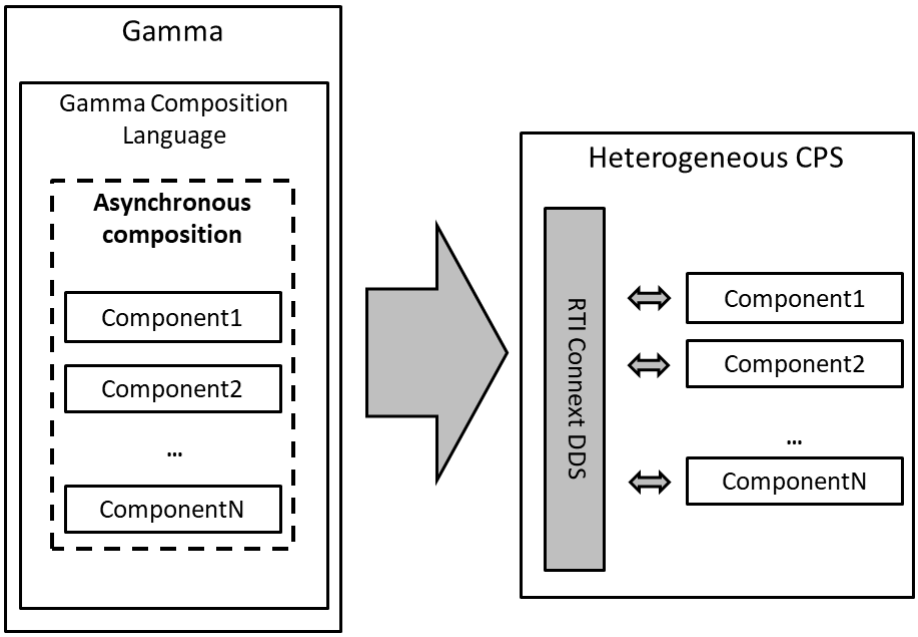

Figure 5.1: Generation of DDS communication for asynchronous components

In addition to the asynchronous composite component, RTI Connext DDS is also required to successfully implement DDS-based communication. The steps of generating communication are the following:

1. An IDL is generated from the composite system, which contains a data type for each channel (port-port connection), system input, and system output in the composition.
2. Publishers and subscribers are generated for each component in the composite system.
3. Using the code generator of RTI Connext DDS, the stubs are generated from the IDL. This creates the necessary files to implement the generated publishers and subscribers.

For each channel, a data type is generated, that is used by the corresponding readers and writers. Components have one publisher and one subscriber. Publishers serve as the output of the component, publishing to topics associated with output channels, while subscribers handle inputs, reading data from topics corresponding to input channels (demonstrated in Figure 5.2). Note that topics associated with system inputs and outputs have no data readers attached to them, or have no data writers writing to them. However, external applications can publish or subscribe to these topics.

The flow of data starts at the subscriber, which upon receiving an update of the data, passes it to the component. The component executes a step with the passed data, then publishes outputs accordingly. Note that although the communication is generated from the composite system, the DDS frame of publishers and subscribers operate with the

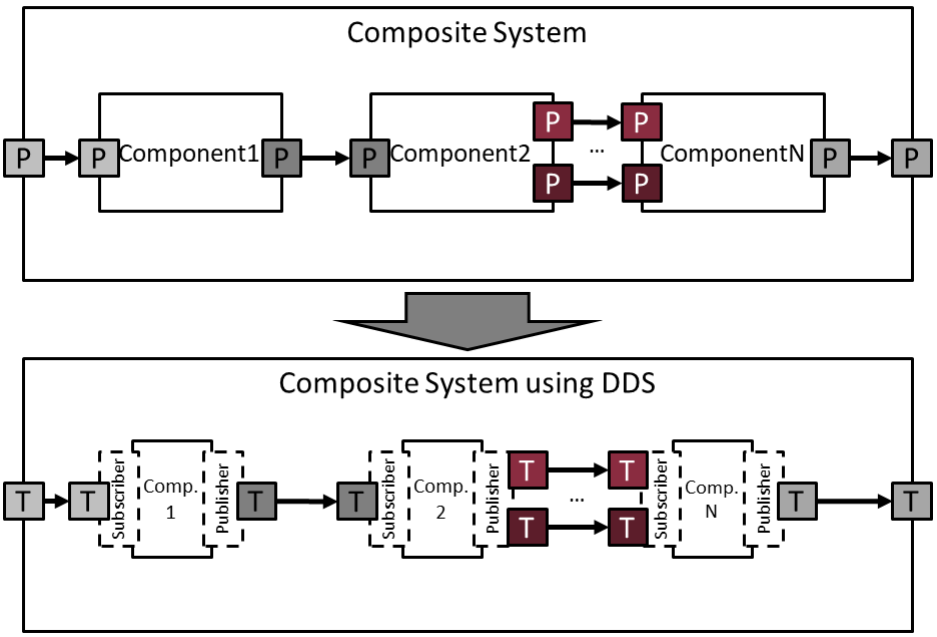

Figure 5.2: Realisation of the DDS implementation *(P stands for port, T stands for topic)*

individual component statecharts, not with the composite system statechart (discussed in Section 4.3.1).

Chapter 6

Implementation

This section presents the code generation tools introduced to Gamma in this work. First, the chapter briefly discusses the necessary capabilities of component implementations in Section 6.1. Then, it presents the notable techniques used when developing the code generators both in C (Section 6.2) and in SystemVerilog (Section 6.3).

6.1 Code Generation

Both C and SystemVerilog code generators reuse architectural and desing patterns from the Java code generator of Gamma. Though the two languages of the new code generators are substantially different, they both have to provide the same functionalities to correctly implement a component.

Each described component has to be able to

- execute a step on the component statechart,
- reset the statechart to the initial state,
- clear input and output events between steps.

The following sections present how the abovementioned points are realised, in addition to clarifying some implementation techniques.

6.2 C Language Implementation

Since the C programming language shares similarities with Java, and the code generator is reuses patterns from the Java generator found in Gamma, I draw comparisons between the two to illustrate the differences and challenges of the C implementation generation.

- In Java, components are described with classes and interfaces. This approach cannot be followed in C, as it does not support object-oriented programming concepts.
- In Java, components have setter and getter functions to manipulate the variables. These are omitted in C, because they present a large overhead.

- In Java, functions that manipulate the state of a component are class member functions, having direct access to class variables. In C, component statecharts structures are passed as a parameter to functions.

Gamma components are described using structs. Upon the generation of the implementation, a struct is created that represents the component, containing all the necessary data elements within. The regions within the component are represented by enum variables. Each literal within an enum represents a state contained in the region. Since C does not operate with namespaces, it is necessary to avoid giving the same name to literals, as the code would not build otherwise. To solve this, each literal has its name extended with the name of the containing region, with a "_" symbol separating the two. For example, the state *Interrupted* within the *Main* region has *Interrupted_main* as a name for its corresponding literal. The generation process ensures that every region has a unique name by prepending the name of each ancestor region (the parent, grandparent, ..., etc. region of the containing state) to the original name of the region (if there is any). Other data elements of the component are represented by basic data types. Inputs and output events are represented with boolean variables, while timeout variables appear as long integers.

```
enum Interrupted {__Inactive____interrupted, Black_interrupted,
    BlinkingYellow_interrupted} interrupted;
enum Normal {__Inactive____normal, Green_normal, Red_normal, Yellow_normal}
    normal;
enum Main_region {__Inactive____main_region, Normal_main_region,
    Interrupted_main_region} main_region;
```

Listing 6.1: Example of enum literal namings

The generation of necessary functions is carried out similarly to the generation of enum literals. Functions that reset or execute steps must be generated for each component, but naming them the same would cause errors. Therefore, these functions have the name of the respective component appended at the end of their name. So the *reset* function of the *TrafficLight* component is called *resetTrafficLight*. In order to use these methods to control the component statecharts, a pointer to an instance of the component needs to be passed as a parameter.

```
void resetTrafficLightCtrl(TrafficLightCtrlStatemachine* statechart);
void changeStateTrafficLightCtrl(TrafficLightCtrlStatemachine* statechart);

void resetController(ControllerStatemachine* statechart);
void changeStateController(ControllerStatemachine* statechart);
```

Listing 6.2: Example of statechart function namings

Since Gamma uses different code generation techniques (described in Section 4.2) that can affect the flow and operation of the generated code, a statechart wrapper is introduced. The wrapper hides the differences between these techniques, while also implementing the timing mechanism used to increment the timeout variables. Generating functions for these wrapper structures is done in the same way as described in the previous paragraph.

The timing of components is currently supported in two ways, since measuring elapsed time is not trivial in C. One approach uses the *gettimeofday* function found in *<sys/time.h>*. This can measure elapsed time in sufficient granularity (milliseconds). However, this cannot be utilised in bare metal scenarios. To solve this, the other method uses clock frequency to measure elapsed time in situations where there is no operating system.

6.3 SystemVerilog Implementation

While the technique used in this code generator reuses patterns from the Java generator as well, the structure of the implementation is vastly different (compared to the previously presented C code generator).

Outputs can be assigned to one or multiple signal values, forming a condition similar to the ones found in *if* statements. If the condition is true, output is observable. Apart from the inputs and outputs of the component, a module also has *clock* and *reset* as input signals. The first is the clock signal produced by the FPGA, and the latter is a trigger to the reset functionality.

```
module TrafficLightCtrlStatemachine(
    clock,
    reset,
    PoliceInterrupt_police_InInput,
    Control_toggle_InInput,
    Lights_displayGreen_Output,
    Lights_displayYellow_Output,
    Lights_displayRed_Output,
    Lights_displayNone_Output,
);

input clock;
input reset;
input PoliceInterrupt_police_InInput;
input Control_toggle_InInput;

output Lights_displayGreen_Output;
output Lights_displayYellow_Output;
output Lights_displayRed_Output;
output Lights_displayNone_Output;


assign Lights_displayGreen_Output = (Lights_displayGreen_Out == 1'b1);
assign Lights_displayYellow_Output = (Lights_displayYellow_Out == 1'b1);
assign Lights_displayRed_Output = (Lights_displayRed_Out == 1'b1);
assign Lights_displayNone_Output = (Lights_displayNone_Out == 1'b1);
```

Listing 6.3: Example of statechart module and input/output generation (*The right hand side of the output assignments consists of the corresponding registers, whose declaration is omitted from the example.)*

Resetting and the changing of states is realised in an *always @ (posedge clock)* block. This block is always executed at the rising edges of the clock signal. This serves as a loop where the state of the component is constantly observed, and if necessary, altered. However, this approach posed some difficulties.

Firstly, while the inputs and outputs of the component are present in the module declaration, they cannot be manipulated inside an *always* block. Verilog (and in turn, SystemVerilog) considers these as *wire* types, and only *reg* (which is short for register) and other basic variables, such as integer variables can be used inside these blocks. As a solution, each input and output of the component has a register generated for them. When an input is raised, its value is stored in the register until the next evaluation of the statechart. This way, inputs can be used in *if* statements in order to change states. Note that the *reset* and *clock* input signals do not have registers generated for them, as it is not necessary. Considering outputs, module outputs are assigned to their respective output registers.

This way, when an output register is set to *1*, the corresponding output is observable. The value of an output register is preserved until the next evaluation.

Secondly, the frequency of the clock in an FPGA is usually much faster than the required operational speed of a component, considering that Gamma supports time values in seconds and milliseconds. This problem bears significance when it comes to timeout events, which fire after the passing of a set amount of time. These events are generated as long integer variables, and should be incremented in the specified time intervals, i.e. every second or millisecond. The timing is solved by having a register named *pps*, which is set to one when the required amount of time passes, based on the clock frequency of the FPGA. This frequency is given as a parameter when the user generates implementation, so the necessary variables are set to the right amount when the code is generated. When the *pps* is set to one, all timeout variables are incremented, and the state of the component is evaluated.

```
always @ (posedge(clock))
  begin
  if (reset) begin
      // The resetting of the statechart happens here.
  end
  if (pps == 1'b1) begin
      //The incrementation of timeout variables and the evaluation of
      //the next state happens here.
  end
end
```

Listing 6.4: The main statechart evaluation loop

The regions within the components are represented by enum variables. Note that this type is not supported by Verilog, only SystemVerilog, making the latter a better choice for code generation in Gamma. Each enum variable is given a namespace, so that literals can have the same name.

```
package Main_regionPackage;
  typedef enum {__Inactive__, Normal, Interrupted} Main_regionEnum;
endpackage
package InterruptedPackage;
  typedef enum {__Inactive__, Black, BlinkingYellow} InterruptedEnum;
endpackage
package NormalPackage;
  typedef enum {__Inactive__, Green, Red, Yellow} NormalEnum;
endpackage
```

Listing 6.5: Example of the namespaces generated for the enums.

Chapter 7

Case Study

This chapter presents case studies that demonstrate the code generation and communication functionalities introduced in this work for the model driven-development of heterogeneous CPS. Section 7.1 introduces the crossroads example, a tutorial example of Gamma in the context of which the case studies are carried out. Section 7.2 presents a case study involving the memory-based communication between an FPGA and a CPU. Then, Section 7.3 presents a case study that uses DDS to establish communication between components.

7.1 The Crossroads Example

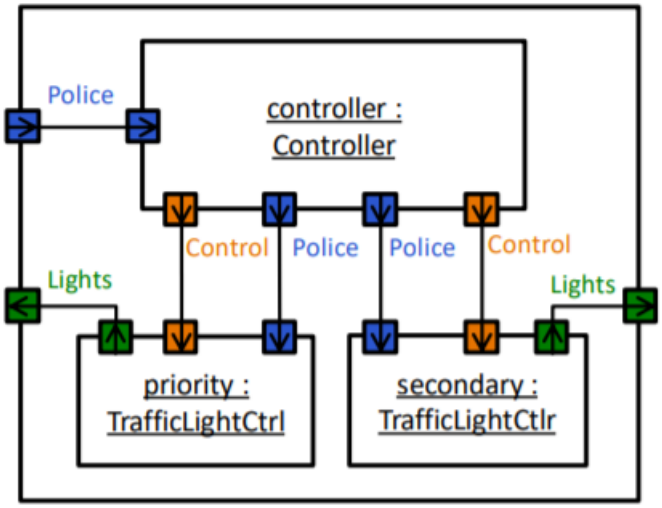

Figure 7.1: The graphical representation of the synchronous composite controller in the crossroads example of the Gamma tutorial

The newly introduced functionalities are presented using the crossroads example of the Gamma tutorial[1]. This example introduces a synchronous composite system that has two traffic lights called *priority* and *secondary* controlling the traffic of the intersecting roads, and a central controller (called *controller*) that controls the traffic lights (the system is demonstrated in Figure 7.1). In the main cycle of the system, the controller (modeled in

[1] https://inf.mit.bme.hu/en/gamma

Figure 7.2) sends toggle signals in defined intervals to the traffic lights via the *Control* ports, causing them to switch between the lights. The active lights of the traffic lights, which can be red, green, yellow or none (denoting the absence of lights), are indicated by signals sent via the Lights ports (the behaviour of the traffic lights is modeled in Figure 4.1). The system can also receive a police signal as input, which is passed to the controller via the *Police* port. Then, the controller forwards this to the traffic lights via the inner *Police* ports, causing them to switch to a blinking yellow state. Another police signal returns the system to the main cycle.

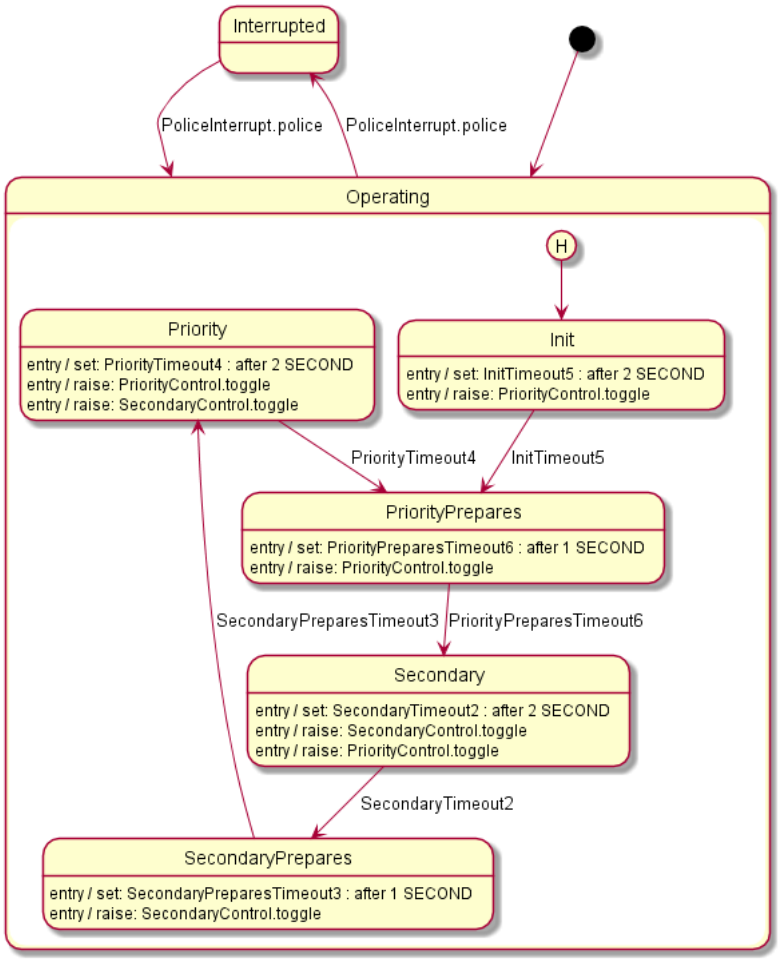

Figure 7.2: The crossroad example

7.2 Crossroads Example on Digilent ZedBoard

This section presents a case study executed on a Digilent ZedBoard[2] (presented in Figure 7.3), a development kit shipped with a Zynq-7000 SoC (a high-level model of the architecture is presented in Figure 3.2).

The system is implemented on a Zynq-7000 SoC (visualised in Figure 7.4). The two traffic lights are synthesized on the Xilinx FPGA, and the controller is executed as software on the CPU.

[2]http://zedboard.org/product/zedboard

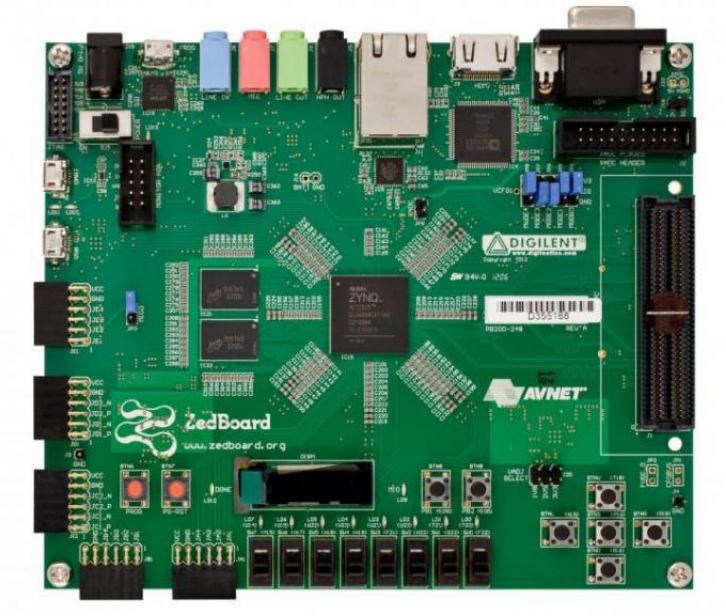

Figure 7.3: A Digilent ZedBoard

7.2.1 Implementing the Traffic Lights

The implementation steps of the traffic lights are the following:

1. SystemVerilog implementation is generated from the Gamma statechart model using the SystemVerilog code generator presented in Section 4.3.2.
2. Using Vivado Design Suite[3], the implementation is packaged as an AXI4 slave peripheral IP.
3. The traffic light AXI IPs are connected to the Zynq-7000 SoC, with outputs connected to the LED GPIOs of the ZedBoard.
4. The design is exported as hardware, serving as target platform for the next step of the case study.

The AXI4 is a high-performance, memory-mapping based interface [29] that handles the sharing of data between the programmable logic and the processing system on the Zynq-7000 SoC. AXI4 operates in accordance with a master-slave protocol, where the master gives the memory address and control to the slave, after which the slave can read data from the master, or the master can write data to the slave.

To utilise the AXI4 interface, the generated traffic light implementation is packaged as an AXI4 slave peripheral IP. During packaging, the traffic light implementation is instantiated inside the AXI wrapper. This is where the user logic gets defined. Since AXI uses memory mapping, and the implementation reads different data values, each input of the traffic light module gets a specific value assigned to them (except for the clock signal). For example, writing 0x000001 to the memory address of any traffic light instance will reset the statechart. Thereafter, the outputs are connected to the LEDs.

[3]https://www.xilinx.com/products/design-tools/vivado.html

By instantiating two traffic light AXI4 IPs, each traffic light gets a memory address, where data sharing can happen with the master. The AXI4 inputs of the traffic lights are connected to the AXI Interconnect, which is connected to the Zynq-7000 system. Each traffic light has 4 LEDs to serve as output indicators, with *red*, *yellow*, *green* and *none* (used in the blinking yellow state to indicate when there are no lights turned on) being the possible outputs.

7.2.2 Implementing the Controller

The controller is implemented on the processing system as software. The implementation steps are the following:

1. The previously designed hardware is imported as target platform in Xilinx Vitis[4].
2. Implementation in C is generated from the Gamma statechart model of the controller using the C code generator presented in Section 4.3.1.
3. The controller is instantiated, gets reset, and is placed in a loop, where it is executed.
4. The system is loaded on the Zynq-7000 SoC.

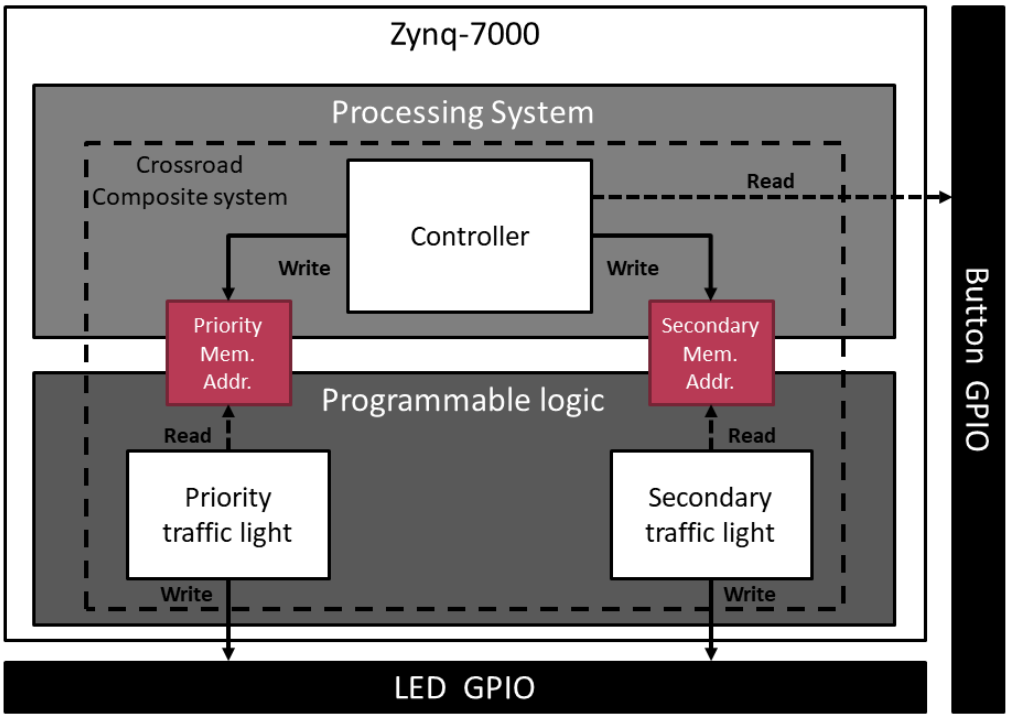

Figure 7.4: The crossroad composite system setup on the Zynq-7000 SoC

The traffic lights are also reset before the loop begins. Inside the loop, a button GPIO is read as well, serving as the police signal. If pressed, the police signal variable of the controller is set. When the controller produces output for any of the traffic lights, the corresponding values are written to the respective memory addresses through the AXI4 peripheral.

[4]https://www.xilinx.com/products/design-tools/vitis.html

The FPGA is programmed with the two traffic lights, while the controller runs on the CPU, that is, on a bare metal environment with no operating system. As a result, the controller produces toggle signals in specific time intervals to the traffic lights, which then light up the LEDs in the right order. If the button gets pressed, the controller signals to the traffic lights, which enter a blinking state. The blinking interval is controlled by the clock of the FPGA.

7.3 Crossroad Example Using DDS

This section presents a case study where the crossroads example (detailed in Section 7.2, visualised in Figure 7.1) is executed in a distributed setting, where individual components are running in different processes and communicate through RTI DDS.

Adhering to the steps described in Section 5.4, the IDL is generated from the crossroad composite component. This contains

- a priority and secondary police type, for the *controller-police* $\longrightarrow$ *priority-police* and *controller-police* $\longrightarrow$ *secondary-police* channels,
- a priority and secondary control type, for the *controller-control* $\longrightarrow$ *priority-control* and *controller-control* $\longrightarrow$ *secondary-control* channels,
- a priority and secondary lights type, for the system outputs of the *priority* and *secondary* traffic lights,
- a controller police type, for the system input of the *controller*.

Then, the publishers and subscribers are generated from the composite component for each standalone component, which publish and subscribe to topics in the same manner as visualised in Figure 5.2. This is followed by the generation of individual statechart implementations from the Gamma models in C language. Finally, using the code generator of RTI Connext DDS, the stubs are generated from the previously derived IDL.

The standalone statechart implementations are placed in a loop, and utilising the corresponding subscribers and publishers, are executed in the following way:

1. when a subscriber receives data, it passes the data values to the statechart implementation,
2. the corresponding variables of the statechart are set, and a step is executed,
3. the outputs are passed to the publisher, which sends them to the subscribers.

This implementation is built and an executable is generated for each component. Upon running these executables, RTI Connext DDS automatically handles the connection of components. The end result is that the *controller* writes data to the priority and secondary toggle topics in set time intervals, which is then received by the subscribers of the *priority* and *secondary* components. The two traffic light components, after processing the input, write output data on the system output topic.

7.4 Summary

The case studies presented in this chapter showcase the capabilities of the novel SystemVerilog, C and DDS code generators. In both case studies, the generated implementations require no further modification in order to be executed. Section 7.2 presents the realisation of the composite component on a heterogeneous target, where one part of the system (the traffic lights) is realised as hardware, and the other (the controller) as software. Section 7.3 presents the same composite system in a distributed setting, where components run as individual processes. Both of these communication implementations are supported by the new data-centric semantic variant, showing the diversity of possible application fields. In Section 7.2, data is conveyed through memory-mapping, between the FPGA and the CPU, while both components run independently, while in Section 7.3, DDS is used to distribute data entries. In both case studies, the C code generated was the same, except for the generated timing mechanism, demonstrating how the new code generators can be used in multiple scenarios. As for the timing in the case study presented in Section 7.3, DDS supports accurate timing mechanisms in C, so utilising this timing for timed events is considered for future works.

Chapter 8

Conclusion and Future Work

In this work I focused on the model-driven development of heterogeneous CPS. CPS often have critical tasks, so exploiting the advantages of model-driven techniques will lead to more reliable systems with less errors. I presented extensions to the Gamma Statechart Composition Framework, supporting the development, verification and testing of heterogeneous cyber-physical systems.

The novel extensions of the Gamma framework are summarized as follows:

- I introduced a new semantic variant for the asynchronous semantics, the so-called data-centric semantic variant, which fits well to the concept of heterogeneous CPS. This novel composition semantics can be used to describe composite systems where data-driven asynchronous communication is used,
- I developed and integrated new code generator techniques to support the automatic derivation of implementation in languages commonly used in embedded systems and real-time systems, and
- I designed an approach to support the networked real-time communication of components and integrated automatic code generators to utilise the efficiency and high performance of DDS.

The applicability of the approaches and functionalities presented in this work is demonstrated in two case studies, each implementing different ways of data-driven interaction.

This is only the beginning of a long road. In the future, I plan to further expand the capabilities of Gamma to support more architectures. To achieve this, the development of new code generators is planned to generate implementation in different languages, such as C++. In addition, I plan to extend the language family of Gamma to support the precise description of the underlying platforms and I plan to integrate optimization techniques to choose the best deployment of the functions to platform elements. As mentioned in Section 2.2, the support of architectural modeling using Gamma is also planned. Furthermore, I also plan to provide a fault-tolerant design pattern library, so the users can choose the level of reliability and availability based on which automatic techniques can synthesize the architectures that satisfy both the functional and extra-functional requirements.

Acknowledgements

I would like to thank my advisors, Bence Graics and dr. András Vörös, who continuously supported me through my work, introducing me to fascinating part of computer engineering, and aiding me professionally.

The results presented in this work were established in the framework of the professional community of Balatonfüred Student Research Group of BME-VIK to promote the economic development of the region. During the development of the achievements, we took into consideration the goals set by the Balatonfüred System Science Innovation Cluster and the plans of the "BME Balatonfüred Knowledge Center", supported by EFOP 4.2.1-16-2017-00021.

Bibliography

[1] Cyber-physical systems - a concept map, 2012. https://ptolemy.berkeley.edu/projects/cps/, Accessed: 2020-10-23.

[2] Health-cps: Healthcare cyber-physical system assisted by cloud and big data. *IEEE Systems Journal*, 2015.

[3] Medical cyber-physical systems: A survey. *Journal of Medical Systems*, 42, 2017.

[4] David F. Bacon, Rodric Rabbah, Sunil Shukla, and T.J. Watson Research Center. FPGA programming for the masses. 2013.

[5] Marco Brambilla, Jordi Cabot, and Manual Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[6] Satrajit Chatterjee, Michael Kishinevsky, and Umit Y. Ogras. xMAS: Quick formal modeling of communication fabrics to enable verification. pages 80–88, 2012.

[7] David Coulter, Mike Jacobs, and Michael Satran. Interprocess communications, 2018. https://docs.microsoft.com/hu-hu/windows/win32/ipc/interprocess-communications?redirectedfrom=MSDN, Accessed: 2020-10-25.

[8] Morin Fleurey. ThingML: A generative approach to engineer heterogeneous and distributed systems. IEEE, 2017.

[9] Thomas Gaska, Marilyn Gaska, Doug Summerville, and Yu Chen. Model based engineering for advanced integrated modular avionics - focus and challenges. 2017.

[10] Leonardo Giordani. Concurrent programming - communication between processes. *LinuxFocus*, 2003. https://tldp.org/pub/Linux/docs/ldp-archived/linuxfocus/English/Archives/lf-2003_01-0281.pdf, Accessed: 2020-10-25.

[11] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling*, 19:1483 – 1517, 2020. DOI: 10.1007/s10270-020-00806-5.

[12] Object Management Group. What is DDS? https://www.dds-foundation.org/what-is-dds-3/, Accessed: 2020-10-21.

[13] Object Management Group. OMG Unified Modeling Language (OMG UML), superstructure, 2009. https://www.omg.org/spec/UML/2.2/Superstructure/PDF, Accessed: 2020-10-23.

[14] Object Management Group. Interface Definition Language. 2018. https://www.omg.org/spec/IDL/4.2/PDF.

[15] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[16] Zoltan Horvat, Velibor Ilic, and Milos Nikolic. Web server and QR decoder applications for Xilinx FPGA boards. 2018.

[17] Real-Time Innovations Inc. RTI Connext DDS—comprehensive summary of QoS policies, 2013. https://community.rti.com/rti-doc/500/ndds.5.0.0/doc/pdf/RTI_CoreLibrariesAndUtilities_QoS_Reference_Guide.pdf, Accessed: 2020-10-21.

[18] Real-Time Innovations Inc. RTI Connext DDS core libraries getting started guide version 6.0.0, 2019. https://community.rti.com/static/documentation/connext-dds/6.0.0/doc/manuals/connext_dds/RTI_ConnextDDS_CoreLibraries_GettingStarted.pdf, Accessed: 2020-10-21.

[19] Vincent Perrier Jean-Paul Calvez. System modeling and architecting with CoFluent Studio, 2005.

[20] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma Statechart Composition Framework: Design, verification and code generation for component-based reactive systems. In *2018 IEEE/ACM 40th International Conference on Software Engineering*.

[21] National Institute of Standards and Technology. Framework for cyber-physical systems: Volume 1, overview. 2017. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1500-201.pdf.

[22] Helen Gill Radhakisan Baheti. Cyber-physical systems. 2011.

[23] Lacey Rae (RTI). DDS databus, 2013. https://community.rti.com/glossary-term/databus, Accessed: 2020-10-25.

[24] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 2003.

[25] Tutorialspoint. Memory mapping, . https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_memory_mapping.htm, Accessed: 2020-10-27.

[26] Tutorialspoint. Inter process communication - pipes, . https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm, Accessed: 2020-10-27.

[27] Tutorialspoint. Shared memory, . https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_shared_memory.htm, Accessed: 2020-10-27.

[28] Remigiusz Wisniewski, Grzegorz Bazydło, Luís Gomes, and Aniko Costa. Dynamic partial reconfiguration of concurrent control systems implemented in FPGA devices. 2017.

[29] Xilinx. AXI reference guide, 2017. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, Accessed: 2020-10-26.