Model-Driven Development of Heterogeneous Cyber-Physical Systems

János Csanád Csuvarszki, Bence Graics, András Vörös
Budapest University of Technology and Economics,
Department of Measurement and Information Systems
Budapest, Hungary
Email: csanad.csuvarszki@outlook.com, {graics, vori}@mit.bme.hu

***Abstract*—Cyber-physical systems (CPS) are becoming more prevalent for the reliable execution of critical tasks, e.g., in the aerospace and medical fields. The development of such systems is challenging, due to the heterogeneity of the components ranging from sensors and embedded controllers to cloud solutions. Model-driven approaches can provide a high-level abstraction to combat the challenges. This paper proposes a model-driven development approach supporting the precise modeling of CPS and the automatic derivation of implementation from the resulting models. The approach introduces a composition semantics tailored to heterogeneous architectures for the precise description of component interactions. Automated code generators allow the execution of standalone software components on real-time controllers and support the synthesis of standalone hardware components, enabling both the derivation of embedded software and the description of logical circuit behavior. Component interactions are supported by multiple communication solutions generally used in critical, real-time embedded systems. The approach is implemented in the Gamma framework and its applicability is demonstrated in two case studies.**

I. Introduction

Cyber-physical systems (CPS) [1] are present in numerous industrial fields, e.g., control systems in automotive, aerospace and healthcare applications. These complex systems not only gather data from their surroundings, but they actively alter the physical world to achieve certain goals. CPS can consist of many heterogeneous components, such as microcontrollers with sensors and actuators, and can connect to processes running on other embedded devices. This heterogeneity makes the development and analysis of CPS a challenge.

This complexity can be 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 code generation for different hardware components is 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. To solve this, we propose a solution in the context of the Gamma framework.

This work was supported by the ÚNKP-20-3 and 4-II New National Excellence Program of the Ministry for Innovation and Technology and the EU ECSEL JU under the H2020 Framework Programme, JU grant nr. 826452 (Arrowhead Tools project) and from the partners' national funding authorities.

The Gamma Statechart Composition Framework [2] is an integrated modeling tool for the semantically well-founded composition and analysis of statechart [3] components. Gamma offers a UML-influenced [4] statechart language to define atomic component behaviour with complex constructs, e.g., orthogonal regions and history states. At its core, it provides a composition language supporting the hierarchical integration of components with precise semantics [5]. Gamma supports system-level formal V&V based on formal models of various model checkers and back-annotating the results. The framework provides test generation functionalities and automated code generators for both statechart and composite models.

In this work, we extend Gamma to support *1)* the modeling of data-centric communication, *2)* the generation of implementation in C and SystemVerilog and *3)* communication between component implementations via DDS or shared memory.

II. Model-Driven Development of CPS

We integrated our model-driven development approach into the Gamma framework in the form of three contribution groups (depicted in Fig. 1), that is, the modeling of composite CPS (Sec. II-A), the derivation of standalone reactive component implementations (Sec. II-B) and supporting the communication of the derived implementation (Sec. II-C). The applicability of our contributions is demonstrated using a running example (Sec. II-D).

A. Modeling Composite CPS

In Gamma, statechart components can be hierarchically composed to create composite components. Components inside such compositions receive inputs and produce outputs through *ports*. These input and output ports can serve as composite system ports, receiving from and producing to outside

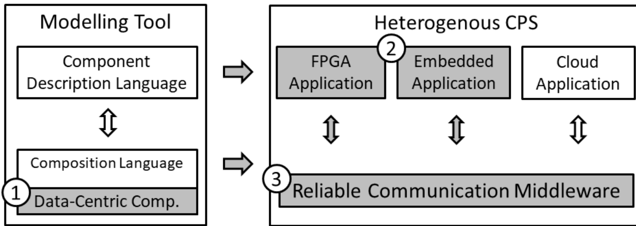

Fig. 1. The steps of our CPS-development approach introduced to Gamma

sources. Components inside a composition use *channels* to connect their ports and communicate with each other.

Contribution 1 introduces a new asynchronous component type, the *data-centric component* to the composition language of Gamma, which uses *data* as a means of communication, a common characteristic of components in heterogeneous CPS. As this paper focuses on the overall model-driven development approach, we give only an overview of data-centric component, and do not include its formal definition.

Data-centric components share data using shared variables, which are modeled by *event declarations* having only *one parameter*: sending and receiving an event with a parameter is equivalent to writing and reading a variable in this case. Currently, each shared variable can have one well-defined *reader* (as a specific event declaration is owned by a single component) and one or more *writers* (other components that can send the specific event via a channel). During execution, if there are multiple writes to the same variable (instances of the same event can be sent multiple times to the same component), always the latest will prevail. This semantic variant does not restrict the running frequency or runtime of components, they run independently of each other.

B. Supporting New Architectures and Platforms

Contribution 2 (detailed in Sec. III) supports the *generation of implementation for embedded controllers and FPGAs* by introducing new code generators to Gamma that rely on the framework's statechart and composition languages to create implementation from state-based models. Currently, Gamma supports the derivation of Java implementation, however, in the case of heterogeneous CPS, there is a need to support code generation for embedded devices. To achieve this, the newly introduced code generators derive implementation in C and SystemVerilog languages. C is one of the most popular languages when developing embedded systems. SystemVerilog is often used in FPGA development, and has high-level language elements that facilitate code generation.

C. Establishing Communication

Contribution 3 (detailed in Sec. IV) introduces RTI DDS, an implementation of the Data Distribution Service standard [6] to Gamma to *establish reliable communication between distributed components*. DDS uses a publish-subscribe model to share data on different topics and operates with a global data space, acting as a local memory that can be accessed via an API. Applications read from and write to these local memories, and the DDS sends messages to update the memories of remote nodes. These local stores supports the applications to the access a global data space. DDS also provides QoS options to increase the reliability of the communication.

To implement the communication between Gamma components, the previously presented C code generator as well as the code generator of RTI DDS is used.

D. Running Example: Crossroads Traffic Controller

We present our approach in the context of a running example involving the design and development of a distributed *crossroads traffic controller* system, presented in the official Gamma tutorial.[1] The crossroads traffic controller system is modeled as a composite component consisting of three atomic statechart components, a *central controller* and two *traffic lights* controlled by the central controller.

The first case study presents *shared memory communication* using a Digilent ZedBoard[2], which consists of a dual-core CPU and a Xilinx FPGA (found inside a Zynq-7000 SoC). The controller runs on the CPU as software, while both traffic lights are synthesised on the FPGA as hardware.

The second case study presents the *distributed communication* of components using DDS. The controller and the traffic lights run in different processes and communicate via RTI DDS. This case study presents our approach both in terms of the operation of standalone HW and SW components and the distributed communication of such heterogeneous components.

III. Modeling and Code Generation for Standalone Hardware

We integrated our modeling and code generation approach into the Gamma transformation chain depicted in Fig. 2.

As an optional step, statechart models created in integrated modeling tools (such as MagicDraw[3] and Yakindu[4]) can be imported by executing model transformations that map these models into the Gamma Statechart Language. This language provides full support to describe self-contained control-oriented, reactive behavior with high-level statechart constructs (e.g., variables, orthogonal regions, history states and complex transitions) and simple action language constructs (e.g., assignments, branches and fixed-iteration loops) and supports data-intensive behavior with manually fillable stubs.

Gamma statecharts can be hierarchically composed according to multiple precise execution and interaction semantics, e.g., asynchronous-reactive, synchronous-reactive and cascade [5]. Both atomic statecharts and composite components can be mapped to models of a low-level transition systems formalism, called EXtended Symbolic Transition Systems (XSTS), serving as input to our introduced code generators.

An XSTS model describes state-based behavior based on *variables* of type boolean, integer or enum, describing system states, and *transitions* specifying possible changes in the state.

[1]The tutorial can be found at http://gamma.inf.mit.bme.hu.
[2]http://zedboard.org/product/zedboard
[3]https://www.nomagic.com/products/magicdraw/
[4]https://www.itemis.com/en/yakindu/state-machine/

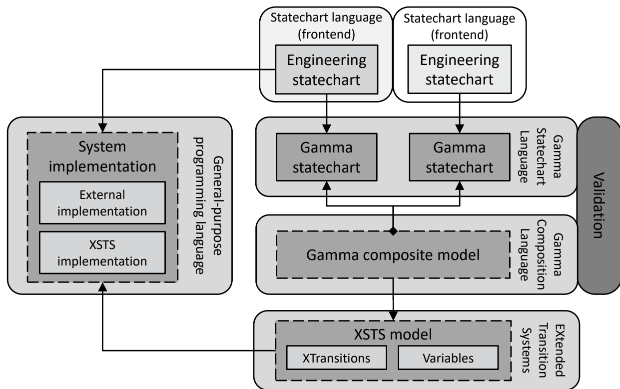

Fig. 2. The model transformation chain for code generation in Gamma

The high-level constructs of (composite) statechart models are mapped to one or more variables. For example, an *event declaration* is transformed into a boolean variable denoting the occurrence of the event and optionally more variables according to its parameters; a *region* is mapped to an enum variable with literals denoting the contained states (which can also be composite). Transitions can use atomic actions, such as assignment actions or assume actions (assumptions that must hold in order to execute specific action branches), which can be reused to construct composite actions, such as sequential actions (i.e., blocks) or deterministic choices.

In our C and SystemVerilog code generators, we map the variables and low-level transition constructs of the XSTS model to constructs of the respective languages (an overview of the whole code generation process is depicted in Fig. 3).

A. C Code Generation

When generating code from a single statechart, multiple source code components are generated. 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 structure that contains the model variables. A wrapper structure is also introduced, which is used to implement timing mechanisms and callback functions. This wrapper contains an instance of the statechart implementation.

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 statechart components, and functions execute operations on the whole system.

B. SystemVerilog Code Generation

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

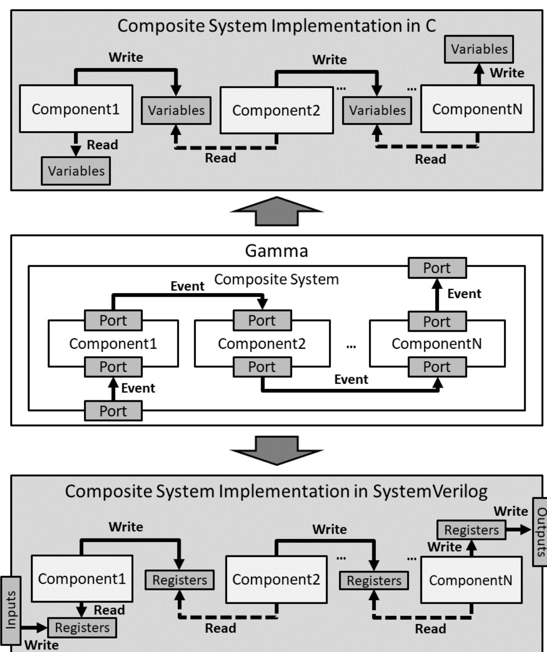

Fig. 3. An overview of the code generation in C and SystemVerilog

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.

C. Example: Digilent ZedBoard

In this example, the implementations of the controller and the traffic lights of the crossroads traffic controller were generated using the C and SystemVerilog code generators.

The traffic light implementations are generated in SystemVerilog from the Gamma statechart model. Then, using Vivado Design Suite, the implementation is packaged as an AXI4 slave peripheral. The traffic light IPs are then connected to the Zynq-7000 SoC and the design is exported as hardware, serving as a target platform for the next step where the controller is implemented. The controller implementation is generated in C from its Gamma model. Then, the controller is instantiated, gets reset, and is placed in a loop, where it is executed. These steps happen in the context of the previously exported target hardware, using Xilinx Vitis. Finally, the system is loaded on the Zynq-7000 SoC, where the controller uses memory mapping to write to the addresses of the traffic lights, signaling them to switch their states.

IV. Code Generation for Distributed CPS

We introduce an RTI DDS code generator for Gamma asynchronous composite components realising the data-centric, distributed communication of composite model implementations.

A. Sharing Data with DDS

Generating the implementation of DDS communication is currently supported in C for asynchronous composite compo-

nents containing data-centric components. In addition to the component implementation, RTI DDS is also required to implement DDS-based communication. First, an IDL (Interface Description Language) file is generated from the composite system, containing a data type for each channel (port-port connection), system input and system output in the composition. Publishers and subscribers are generated for each component in the composite system. Finally, using the code generator of RTI DDS, stubs are generated from the IDL file, creating the files to implement the generated publishers and subscribers.

For each channel, a data type is generated, which is used by the corresponding readers and writers. Components have one publisher and one subscriber. Publishers serve as the output of the component, while subscribers handle inputs (Fig. 4 presents an overview of this).

B. Example: RTI Connext DDS

To realize DDS communication , the statechart implementation wrapper of the component is placed in a loop, where it is repeatedly executed. When its subscriber receives data, it passes it to the statechart implementation. The statechart processes the input, executes a step, and if possible, produces an output. The output is passed to the corresponding publisher, which publishes the data to other component subscribers.

The implementation of the controller and traffic light components is generated via the introduced C code generator. Then, from the composition, the necessary files are generated to establish communication via DDS. From the generated IDL, the necessary stubs are created using the code generator of RTI DDS. Finally, the project is built, and the processes of components are executed. As a result, the components communicate via RTI DDS, behaving according to their statechart model.

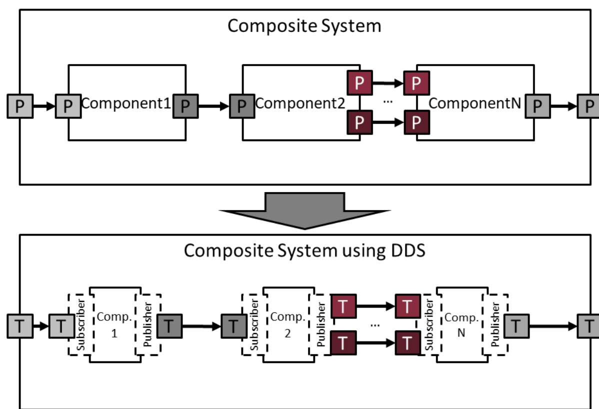

Fig. 4. An overview of the implementation of DDS communication (P stands for port, T stands for topic)

V. Related Work

This section presents solutions addressing the model-driven development of heterogeneous CPS that provide similar approaches in modeling and generating implementation.

ThingML In [7], 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 component behaviour and supports asynchronous communication while supporting the generation of implementation and communication. Contrary to Gamma, ThingML does not support the semantically sound mix-and-match composition of components according to various execution and interaction semantics (synchronous and asynchronous), code generation for hardware description languages and formal verification.

xMAS In [8], a set of microarchitectural primitives is defined, allowing 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. The article proposes an approach based on creating executable and analyzable high-level models, called xMAS (eXecutable Microarchitectural Specification) models. For formal verification purposes, it generates Verilog out of the models and uses inhouse and academic tools. This approach focuses on defining models on a microarchitectural level, rather than defining high-level behaviour models, like Gamma.

VI. Conclusion and Future Work

In this paper, we presented extensions to the Gamma framework that enable the model-driven development of heterogeneous CPS. We introduced a *data-centric communication semantics* for the precise description of communication between modeled components, *code generators* to support the development of multiple embedded platforms as well as a *middleware-based communication* solution to enable the distributed functioning of components.

In the future, we plan to extend the architectural modeling capabilities of Gamma. By defining the target platforms, the framework could allocate components to the specified targets, automatically creating corresponding implementation.

References

[1] N. I. of Standards and Technology, “Framework for cyber-physical systems: Volume 1, Overview,” 2017, http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1500-201.pdf.

[2] V. Molnár, B. Graics, A. Vörös, I. Majzik, and D. Varró, “The Gamma Statechart Composition Framework: design, verification and code generation for component-based reactive systems,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 113–116.

[3] D. Harel, “Statecharts: A visual formalism for complex systems,” *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[4] “OMG Unified Modeling Language (OMG UML), Superstructure,” pp. 525–582, 2009, https://www.omg.org/spec/UML/2.2/Superstructure/PDF.

[5] B. Graics, V. Molnár, A. Vörös, I. Majzik, and D. Varró, “Mixed-semantics composition of statecharts for the component-based design of reactive systems,” *Software and Systems Modeling*, vol. 19, pp. 1483 – 1517, 2020.

[6] “What is DDS?” https://www.dds-foundation.org/what-is-dds-3/.

[7] F. Fleurey and B. Morin, “ThingML: A generative approach to engineer heterogeneous and distributed systems,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 185–188.

[8] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, “xMAS: Quick formal modeling of communication fabrics to enable verification,” *IEEE Design Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.