The Gamma Statechart Composition Framework

Design, Verification and Code Generation for Component-Based Reactive Systems

Vince Molnár[1,2], Bence Graics[1], András Vörös[1,2], István Majzik[1], Dániel Varró[1,2,3]
[1] Fault Tolerant Systems Research Group, Budapest University of Technology and Economics
[2] MTA-BME Lendület Cyber-physical Systems Research Group
[3] Department of Electrical & Computer Engineering, McGill University
{molnarv,vori,majzik,varro}@mit.bme.hu

ABSTRACT

The Gamma Statechart Composition Framework is an integrated tool to support the design, verification and validation as well as code generation for component-based reactive systems. The behavior of each component is captured by a statechart, while assembling the system from components is driven by a domain-specific composition language. Gamma automatically synthesizes executable Java code extending the output of existing statechart-based code generators with composition related parts, and it supports formal verification by mapping composite statecharts to a back-end model checker. Execution traces obtained as witnesses during verification are back-annotated as test cases to replay an error trace or to validate external code generators.

Tool demonstration video: https://youtu.be/ng7lKd1wlDo

CCS CONCEPTS

• **Theory of computation → Verification by model checking**; • **Software and its engineering → System modeling languages**; **Formal software verification**;

KEYWORDS

statecharts, composition, formal verification, code generation

1 INTRODUCTION

Statecharts are a popular [1, 2] language to capture the behavior of reactive systems [3] that react to external stimuli depending on their internal state. Statecharts provide an expressive formalism to represent complex state-based behavior by introducing hierarchical state refinement, memory (variables and history state) and complex transitions (e.g., fork and join transitions). As statecharts have become part of key industrial modeling standards like the Unified Modeling Language (UML) or SysML, there are a large number of design tools supporting (different variants of) the formalism such as MagicDraw, BridgePoint, or Rhapsody. Moreover, there are also specialized industrial tools with emphasis on customizable code generation such as Yakindu Statechart Tools from Itemis[1].

[1]https://www.itemis.com/en/yakindu/state-machine/

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183489

In this paper, we present the Gamma Statechart Composition Framework which aims to fill the gap between enterprise-level UML tools and specialized statechart tools by providing a layer for composing individual statechart components while extending the capabilities of automatic code generation and verification and validation (V&V). Thus the *intended target audience of the Gamma framework* includes all software engineers who are working with industrial statechart modeling tools.

As major contributions, Gamma *1)* provides a modeling language and framework for the *hierarchical composition* of heterogeneous off-the-shelf statechart components in an object-oriented manner (see Section 2.1), *2)* integrates a 3rd party statechart modeling tool and model checker to support the formal verification of composite system models (Section 2.3) and *3)* automatically generates an implementation of the composition code on top of existing auto-generated source code of individual components (Section 2.2) as well as test cases for validation (Section 2.4). As a showcase, the framework is currently integrated (as a front-end) with the Yakindu Statechart Tools as an off-the-shelf statechart modeling and code generation tool, and (as a back-end) with UPPAAL [4], a model checker for timed automata to provide the verification capabilities.

While there are tools with similar goals (see Section 3), the main added value of Gamma is to uniquely combine and integrate the strength of off-the-shelf statechart and verification tools thus providing an end-to-end solution for statechart based compositional design, formal verification and validation as well as code generation. Existing integrative approaches are either restricted to focus only on the statechart model of a single component of the system (e.g., [5–7]), or the composition semantics for building complex systems from components cannot be efficiently mapped to a formal verification and validation framework [8]. Finally, enterprise-level UML tools often support compositional modeling, but formal verification and validation is rarely available and often difficult to use.

2 FEATURES OF GAMMA

The features of Gamma (see Figure 1) will be presented on an example of controlling traffic lights at a crossroad where one statechart describes a single traffic light (*light controller*), and another captures the synchronization of the two directions (*crossroad controller*). Gamma allows engineers to compose a system from components, synthesize source code as implementation, verify if safety requirements are satisfied, and generate test cases for validation.

2.1 Modeling Hierarchical Statechart Networks

Gamma offers the *Gamma Composition Language (GCL)* to describe components, interfaces and ports, and communication channels.

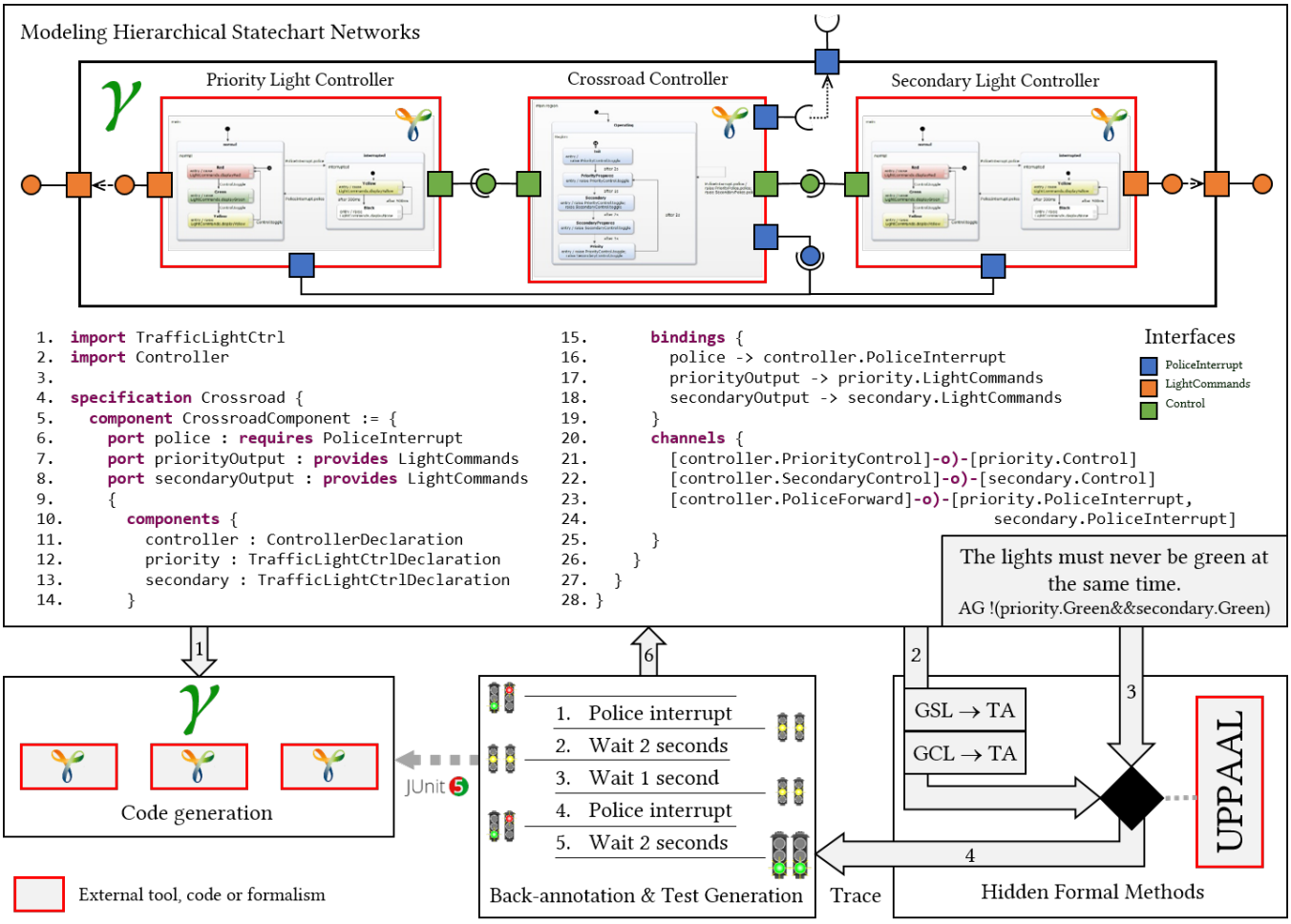

Figure 1: Overview of the features of the Gamma Statechart Composition Framework.

Currently, there are two types of components in GCL. The basic building block is a *statechart component*, which is a single statechart model described in the *Gamma Statechart Language (GSL)*. This language serves as an intermediate modeling formalism that enables the integration of external modeling tools. Yakindu Statechart models are first compiled to GSL including all the mappings of Yakindu interfaces to Gamma interfaces and ports. Then *Composite components* are defined in GCL as the composition of other components together with their respective ports, and realized interfaces. Such composition includes *1)* the instantiation of constituent components, *2)* the definition of port bindings, (i.e., a mapping of the ports of the composite component to the ports of the constituent components), and *3)* the definition of communication channels.

In GCL, *components communicate via ports* where each port defines a point of service where certain signals can be sent or received. A signal is a piece of data passed between components potentially with a parameter. Signals are declared on interfaces, which may be realized by ports. A signal may be declared as *input*, *output* or *in/out*, which means that it can be received, sent or both through the realizing port in case of a *provide* interface (while these directions are reversed in case of *require* interfaces). A *broadcast interface* is a special type of interface on which every signal is *output*.

Communication is carried out through channels. *Simple channels* can connect two ports if they implement the same interface but in different modes, i.e., the signal directions will be exactly the opposite on the two ports. *Broadcast channels* allow a single port *providing a broadcast interface* to be connected to multiple ports *requiring the same broadcast interface*. To avoid race conditions, GCL disallows a single port to be connected to more than a single channel, when multiple signals could arrive to the same port at the same time, thus nondeterministically overwriting each other.

In the crossroad example (top of Figure 1), the light controllers have three ports – one for the toggling the state of the traffic light, one for the police to force the light into a blinking yellow state, and one for the output signals for the traffic light hardware. The most important signal (for toggling the state of the lights) is defined on the *Control* interface, which is realized in *required* mode by the ports of the light controllers, and in *provided* mode on the crossroad controller which will generate the signals. These ports are connected by simple channels, while the police interrupt signal

is broadcast to the light controllers. The composite crossroad system has three ports, one for the police (mapped to the corresponding port of the crossroad controller) and two for the outputs of the light-controllers, mapped to the output port of one of the two instantiated light controllers (priority and secondary).

The Gamma Composition Language has a formally defined execution semantics based on *synchronous-reactive* composition semantics [8]. Components are scheduled by a periodic external trigger to start an execution cycle. In each cycle, each component is executed exactly once while processing their inputs and producing their outputs, while potentially changing their internal state. The execution order is irrelevant since a signal sent in a cycle will only arrive in the next cycle. The restriction on channels ensures that no race condition will occur inside the system by sending more than one signal to the same component within the same cycle.

2.2 Code Generation

Once the entire system is modeled as a hierarchical network of communicating statechart components, Gamma can generate the implementation of the composition code on top of existing auto-generated source code of individual components. External code generators for statechart components can be integrated by implementing a plugin for the composition code generator, wrapping the external generated code behind the interfaces generated by Gamma. The framework also provides test cases to validate that the external code generation conforms to the intermediate Gamma statechart model (see Section 2.4).

Gamma currently supports the generation of Java source code. The interfaces defined in GCL are translated into Java interfaces and accompanying listeners which enables to integrate system signals and actuators with the generated controller. The system as a whole is enclosed in a separate class so that system-level ports (implementing the interfaces) can be reached and an execution cycle can be invoked. The external invocation of the execution provides a way to fine-tune the behavior of the system. For example, the crossroad system is invoked periodically, but it is also possible to execute the system every time an input signal arrives.

2.3 Hidden Formal Methods

A key design goal of Gamma is to extend the formal verification of single statechart (e.g., [5–7]) to the verification of complex systems built up from these individual statecharts as components, which is rarely supported in practice. Our strict formal compositional semantics of synchronous reactive behavior allows efficient model checking for verification since no interleaving of component executions needs to be dealt with. Essentially, executions of Gamma models are deterministic with respect to external input, but the environment producing the input is modeled as non-deterministic.

To assist software engineers, Gamma hides the inherent complexity of using formal methods by offering a high-level user interface for verification, and by exploiting automated model transformations and back-annotation of verification results [9].

Model checking based formal verification necessitates to provide a formal property to be checked. Most verification tools expect to capture such properties as a temporal logic expression, which requires significant expertise to write [10]. In Gamma, we have followed [11] to define textual templates for the supported temporal operators of the underlying model checker tool, UPPAAL. These templates have placeholders to be filled by logical expressions over system states constituted from the states of components and their variables. Templates also come with a typical sample property that is usually described with the specific temporal operator.

For example, a safety property of the crossroad system may be captured by the *"Must always"* template: *"The model must always satisfy the following condition during every behavior"* (for which the sample is: *"A critical error must never occur."*). In our context, the crossroad system should never get into a state where both traffic lights are green (thus, the logical expression in the template is "!($priority.Green \&\& secondary.Green$)"). Unfortunately, the example model does not satisfy this requirement.

2.4 Back-Annotation and Test Generation

Once the formal model and the property of interest are available, Gamma can execute the back-end verification tool (UPPAAL) and display the results, which is either the satisfaction of the property or a counter-example provided by the model checker as a witness that proves or refutes the satisfaction of the property. To help engineers understand and fix the discovered problem, execution traces retrieved by the model checker are back-annotated to the high-level statechart models.

Gamma is capable of back-annotating witnesses as a sequence of delays (to recover the timed behavior), external inputs, expected states and expected outputs. This can be used to evaluate the behavior, but simulation in the Yakindu Statechart Simulator is not yet supported, because Yakindu currently supports the simulation of a single statechart model only. Fortunately, Gamma also generates a jUnit test suite that replays the sequence on the generated implementation to check if the system reacts in the expected way.

In addition to visualizing a witness, the auto-generated jUnit test cases can also be used to *validate external code generators* (e.g., that of Yakindu Statecharts) wrt. a designated set of properties. As there is no guarantee that the external code generator fully conforms to the interpreted behavior of the translated Gamma statechart model, we use the concepts of model-based testing [12] to enforce the model checker to generate execution traces to achieve a designated test coverage. Such a coverage criterion can be that each state of every statechart should occur in at least one (system-level) state in the generated traces. With this test suite, designers can gain a certain level of confidence in the correctness of the framework, and most importantly, detect potential problems before deploying the generated code into a critical environment.

In our example, the controller and the traffic light may fall out of sync if the police interrupts normal operation (see the trace in Figure 1). Back-annotating the trace retrieved by the model checker can reveal the actual scenario that violates the safety property.

3 RELATED TOOLS

We provide a detailed feature comparison (in Table 1) with popular tools which support statecharts, composition and/or verification/validation: Ptolemy II, BIP, MATLAB/Simulink Stateflow (SL/SF) and IBM Rational Software Architect Realtime Edition (RSARTE).

	Native statecharts	External editors	Synchronization-based	Event-based	Code generation	Simulation	Formal verification	Test generation	Extensibility	Free to use
Gamma	✓	✓		✓	✓		✓	✓	✓	✓
Ptolemy II			✓	✓	✓̸	✓	✓̸		✓	✓
BIP		✓	✓		✓	✓	✓		✓	✓
SL/SF	✓			✓	✓	✓		✓		
RSARTE	✓	✓		✓	✓	✓			✓	

Table 1: Features of Gamma and its competitors. ✓= fully supported; ✓̸= experimental

Ptolemy II[2] [8] is an open-source modeling framework that supports the modeling and simulation of hierarchical composite systems with various component implementations and interaction semantics. Components in Ptolemy II are based on actors, which are custom programs. Their interaction semantics is defined by *directors* which define a model of computation along different hierarchy levels. The different directors (e.g., rendez-vous, synchronous-reactive or discrete events) can be combined on different hierarchy levels, and even more complex behavior can be achieved by the use of *modal models* (where the activation of actors are controlled by a state machine). Ptolemy II offers simulation capabilities for its rich modeling languages, but its former code generator module has been discontinued and it does not offer formal verification capabilities.

BIP[3] [13] (Behavior, Interaction, Priority) is a modeling framework focusing on component interactions. BIP defines a powerful language to define interactions, but contrary to Gamma, these interactions are synchronization-based, coupled with the description of data-flow. BIP offers a rich toolset, containing several transformers for third-party models (e.g., MATLAB/Simulink, or AADL), code generators to produce C/C++ or Java code, and supports the formal verification of invariant properties and deadlock-freedom.

Stateflow[4] [14] is a commercial framework for the design of reactive (embedded) systems. Stateflow supports the hierarchical modeling of composite statechart systems using various scheduling algorithms, and it can simulate and validate the models and generate source code. Stateflow is a very mature tool with professional support and rich features, but it offers restricted support for 3rd party extensions and commercial licenses are expensive.

RSARTE[5] is a commercial Eclipse-based modeling framework. Components are called capsules and RSARTE defines the communication among the capsules through ports. Composition and hierarchy is provided by using composite structure diagrams, which enables the hierarchical refinement of systems. RSARTE generates code from the design models and uses model simulation and Eclipse CDT to test and debug the model and its implementation.

[2]http://ptolemy.eecs.berkeley.edu/
[3]http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=en
[4]https://www.mathworks.com/products/stateflow.html
[5]https://www.ibm.com/support/knowledgecenter/SS5JSH/rsart_family_welcome.html

4 APPLICATIONS

The key benefits of Gamma, i.e., an intuitive but precise composition language, the integration of Yakindu Statechart Tools, the use of hidden formal methods and the automated generation of tests, has been demonstrated at various academic and industrial venues.

The authors have used Gamma as an educational tool in a thematic laboratory at Budapest University of Technology and Economics, Hungary, and in an undergraduate course at McGill University, Canada. It was also used as part of the 8th Summer School on Domain Specific Modelling Theory and Practice[6] (DSM-TP'17).

The embedded safety controller logic of the MoDeS project[7] has been designed and verified using Gamma. The project won the 3rd prize at the international Eclipse IoT Developer Challenge in 2016. The Gamma framework was presented to a large industrial audience at EclipseCon Europe in 2017. Gamma and a detailed tutorial (including the example used in this paper) are available at http://gamma.inf.mit.bme.hu.

ACKNOWLEDGMENTS

This paper was partially supported by the ÚNKP-17-2-I and ÚNKP-17-3-I New National Excellence Programs of the Ministry of Human Capacities. Special thanks to Tamás Tóth for his initial contributions to the Gamma Statechart Language and István Ráth and Ákos Hajdu for their valuable advice.

REFERENCES

[1] G. Reggio, M. Leotta, and F. Ricca, *Who Knows/Uses What of the UML: A Personal Opinion Survey.* Cham: Springer, 2014, pp. 149–165.
[2] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, *Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain.* Springer, 2014, pp. 166–182.
[3] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987.
[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, Y. Wang, and C. Weise, "New Generation of Uppaal," in *WS on Software Tools for Technology Transfer*, 1998.
[5] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker," *Formal Aspects of Computing*, vol. 11, pp. 637 – 664, 1999 1999.
[6] Y. Jiang, Y. Yang, H. Liu, H. Kong, M. Gu, J. Sun, and L. Sha, "From stateflow simulation to verified implementation: A verification approach and a real-time train controller design," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–11.
[7] Y. Meller, O. Grumberg, and K. Yorav, *Verifying Behavioral UML Systems via CEGAR.* Springer International Publishing, 2014, pp. 139–154.
[8] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.
[9] Á. Hegedüs, G. Bergmann, I. Ráth, and D. Varró, "Back-annotation of simulation traces with change-driven model transformations," in *8th IEEE Int. Conf. on Software Engineering and Formal Methods, SEFM 2010*, 2010, pp. 145–155.
[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE 1999*, 1999, pp. 411–420.
[11] B. F. Adiego, D. Darvas, J. Tournier, E. B. Viñuela, and V. M. G. Suárez, "Bringing automated model checking to PLC program development - a CERN case study," in *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014.*, 2014, pp. 394–399.
[12] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Softw. Test., Verif. Reliab.*, vol. 22, no. 5, pp. 297–312, 2012.
[13] M. D. Bozga, V. Sfyrla, and J. Sifakis, "Modeling synchronous systems in BIP," in *Proc. 7th ACM Int. Conf. on Embedded Software*, ser. EMSOFT '09, 2009, pp. 77–86.
[14] S. T. Karris, *Introduction to Stateflow with Applications.* Orchard Publ., 2007.

[6]http://msdl.cs.mcgill.ca/conferences/dsm-tp-2017
[7]https://inf.mit.bme.hu/research/projects/modes3