Software and Systems Modeling (2020) 19:1483–1517
https://doi.org/10.1007/s10270-020-00806-5

THEME SECTION PAPER

Mixed-semantics composition of statecharts for the component-based design of reactive systems

Bence Graics[1,2] · Vince Molnár[1,2] · András Vörös[1,2] · István Majzik[1] · Dániel Varró[1,2,3]

Received: 16 June 2019 / Revised: 5 May 2020 / Accepted: 7 May 2020 / Published online: 1 July 2020

Abstract
The increasing complexity of reactive systems can be mitigated with the use of components and composition languages in model-driven engineering. Designing composition languages is a challenge itself as both practical applicability (support for different composition approaches in various application domains), and precise formal semantics (support for verification and code generation) have to be taken into account. In our Gamma Statechart Composition Framework, we designed and implemented a composition language for the synchronous, cascade synchronous and asynchronous composition of statechart-based reactive components. We formalized the semantics of this composition language that provides the basis for generating composition-related Java source code as well as mapping the composite system to a back-end model checker for formal verification and model-based test case generation. In this paper, we present the composition language with its formal semantics, putting special emphasis on design decisions related to the language and their effects on verifiability and applicability. Furthermore, we demonstrate the design and verification functionality of the composition framework by presenting case studies from the cyber-physical system domain.

Keywords Component-based design · Statecharts · Composition language · Formal semantics · Formal verification

Communicated by Federico Ciccozzi, Antonio Cicchetti and Andreas Wortmann.

✉ Bence Graics
graics@mit.bme.hu

Vince Molnár
molnarv@mit.bme.hu

András Vörös
vori@mit.bme.hu

István Majzik
majzik@mit.bme.hu

Dániel Varró
varro@mit.bme.hu

[1] Deptartment of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[2] MTA-BME Lendület Cyber-physical Systems Research Group, Budapest, Hungary

[3] Department of Electrical and Computer Engineering, McGill University, Montreal, Canada

List of symbols

Tuples, sets and sequences

$(\ldots)$	Tuple
$\{\ldots\}$	Set
$\lvert\ldots\rvert$	Size of set
$\langle\ldots\rangle$	Sequence
A^*	The set of finite (possible repeating) sequences of the elements of the set A
ε	The empty sequence
$\lvert s\rvert$	The length of sequence s
$s[i]$	The ith element of sequence s
$\sigma(a)$	A permutation of set a
$S_\sigma(a)$	All permutations of set a

Indices and sizes

i	A generic index variable
j	A secondary generic index variable
a'	Another value for variable a
n	A generic size variable
k	An index used for components
K	The number of components

Temporal valuations

a'	Next value of a
$a^{(i)}$	The value of variable a in the ith step

Component types

⊖	Definition of synchronous components ("single-threaded")
$⊖_k$	The kth synchronous component
ⓢ	Definition of synchronous composites
ⓒ	Definition of cascade composites
?⃝⟩⊖	Definition of composite ?⃝ as a synchronous component
⊜	Definition of asynchronous components ("multi-threaded")
$⊜_k$	The kth asynchronous component
⊏⃝	Definition of the asynchronous wrapper component
ⓐ	Definition of asynchronous composites
?⃝⟩⊜	Definition of ?⃝ as an asynchronous component
a_k	Part a of the kth component (e.g., S_k of $⊖_k$)

Events

e	An event
E	The set of events
I	The set of input events
O	The set of output events
$\mathcal{D}$	The domain function
$\mathcal{D}(e)$	The domain of event e
d	An element of a domain
d_i	The ith element of a domain
p	A parameter (of an event)
$\perp$	The empty parameter denoting that an event is missing
$inst(e)$	The instances of event e
$inst_{\perp}(e)$	The instances of event e including the "null" instance $(e, \perp)$
$\hat{I}$	All input events of constituent components
$\hat{O}$	All output events of constituent components
$\hat{\mathcal{D}}$	All domains of events in constituent components

Event vectors

v_E	An event vector for events in E
V_E	All possible event vectors for events in E
v_I	An input vector
v_O	An output vector
V_I	All possible input vectors
V_O	All possible output vectors
$v_{\hat{O}}$	The last output of constituent components
$V_{\hat{O}}$	All possible last outputs of constituent components
$\perp_{\hat{O}}$	Empty output vector for all constituent components
$\perp_I$	Empty input vector

Event sequences

q	A queue (a sequence of event instances)
ω	An output sequence of event instances
Ω	The set of possible output sequences of event instances

Component parts

S	The set of states
s	A state
s^0	The initial state
T	A state-transition function
t	A single state-transition

Composite parts

$\mathbf{C}$	Constituent components of a composite
$\rightleftharpoons$	Channels in a composite
$\rightleftharpoons(e)$	The events linked to event e determined by the channels
X	The order of execution of components of cascade composites

Asynchronous adapter

e_c	The control event
C	The set of clocks in an adapter
$c_i^{t_i}$	A clock producing e_c periodically after every t_1
$trig$	The trigger predicate
a_s	Part a of the wrapped synchronous component (e.g., S_s of $⊖_s$)

Messages

m	A message
e_O	The source event of a message
E_O^{ext}	The output event set of the environment
e_I	A single target event of a message
E_I	Target events of a message
E_I^{ext}	The input event set of the environment
$src(m)$	The source (sending) component of m
ⓔ	The environment

Occurrences

$send(m)$	The occurrence of sending message m
$recv(m, e_I)$	The occurrence of receiving message m on target event e_I
$[t]$	An occurrence of transition t
m_I	A message triggering the occurrence of a transition
m_O	A message generated by an occurrence of a transition

M_O	The messages generated by an occurrence of a transition
$\#occ$	The position of occurrence *occ* in a total ordering of occurrences
$\tau(occ)$	The timestamp of occurrence *occ* according to a global clock

1 Introduction

Reactive systems process events coming from the environment and react to them in accordance with their internal states. In the case of complex reactive systems, platform- or component-based design techniques [1] can be used that integrate reactive components, possibly in a hierarchical way. The implementation and verification of the composite system necessitate precise design languages both at component level and integration level.

To model the dynamic behavior of reactive components, statecharts [2] is an expressive language offering powerful constructs, such as composite states for hierarchical state refinement, parallel regions for describing parallel behavior, history states and variables for expressing memory, and choices, fork and join transitions to model complex transitions and actions. Modeling standards like UML and SysML have adopted the statechart formalism, but they intentionally left parts of its dynamic semantics formally un- or under-specified [3,4], even in the recently published specifications (fUML [5], PSSM [6] and PSCS [7]). These specifications provide execution semantics, but several aspects are not fixed (e.g., there are no assumptions about global synchronization). As a result, design tools, such as Rhapsody,[1] BridgePoint,[2] MagicDraw,[3] and Yakindu Statechart Tools[4] support slightly different semantic variants of the formalism [8,9].

At the integration level, existing modeling standards support the (de)composition of components on a rather syntactic level. The Component Diagram and the Composite Structure Diagram of UML, and the Internal Block Diagram of SysML provide support to capture what type of interaction (data or control) is intended between the components, but they do not provide precise behavioral semantics to allow direct implementation and formal verification of the composite system (i.e., the exact behavior resulting from component interactions, particularly related to the scheduling of components). Thus, the composition of individual components in a well-founded way is a significant challenge for system integrators, who may even have to cope with the problem that components provided by vendors may originate from different modeling tools with slightly different concrete syntax and semantic variants.

Our Gamma Statechart Composition Framework [10] is an integrated modeling tool that aims to support the semantically well-founded composition of heterogeneous statechart components where individual components may use different statechart semantics. The Gamma framework intentionally reuses statechart models of existing tools and their respective code generators for individual components. At its core, it provides the Gamma Composition Language that supports the interconnection of components in a hierarchical way on the basis of precise semantics, *including scheduling strategies and constraints*. Additionally, Gamma provides automated code generators for composite models and test case generators for the interaction between components. Gamma also supports system-level formal verification and validation (V&V) by mapping statechart and composition models into formal models of the UPPAAL[5] [11] model checker.[6]

In this paper, we focus on the Gamma Composition Language and present its elements with their formal syntax and design examples. We also present the formal semantics of the language, discussing the related design decisions that coped with practical applicability and verifiability of the composed system and allowed the implementation of the above-mentioned code generation, formal verification, and test case generation functionalities. The main novelty of our language is that it (1) expects a very simple reactive behavior and interfaces from the system components facilitating component heterogeneity, (2) supports three variants of composition semantics, namely asynchronous-reactive, synchronous-reactive, and cascade, covering a notable portion of execution modes used in reactive systems, and (3) allows the mixing of these composition semantics in a hierarchical way. To demonstrate the use of the composition language and the related verification functionality, we present case studies from the critical cyber-physical system domain.

The paper is structured as follows. Section 2 presents the motivations and the basic concepts related to the Gamma framework. The main contribution of the current work, that is, the Gamma Composition Language (and its related languages for interfaces, expressions and test cases) is presented in Sect. 3, whereas the formal semantics of the composition language is introduced in Sect. 4. Section 5 summarizes how this semantics supported the functionalities of the framework. The applicability of the Gamma framework is demonstrated through three case studies. Section 6 demonstrates the applicability of the languages of Gamma for the description of SysML behavioral models. Section 7 demonstrates the usability of Gamma in the formal analysis of a communication protocol considering physical phenomena

[1] https://www.ibm.com/us-en/marketplace/rational-rhapsody/.

[2] https://xtuml.org/.

[3] https://www.nomagic.com/products/magicdraw/.

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

[5] http://www.uppaal.org/.

[6] The tool can be extended to map to other formalisms.

and failure modes in the communication between protocol participants. Section 8 presents an abstraction-based verification approach in Gamma in the context of a railway-themed case study. The case studies are followed by the presentation of related modeling languages, tools and frameworks in Sect. 9. Finally, Sect. 10 provides concluding remarks and plans for future work.

2 Preliminaries

This section starts with a motivating example (Sect. 2.1), which is used throughout the paper for presenting various language constructs. Next, the composite reactive modeling semantics supported by our composition language are introduced (Sect. 2.2). The section ends with an overview on the functionalities of the Gamma framework (Sect. 2.3).

2.1 Running example

We demonstrate the development of component-based reactive systems in the Gamma framework in the context of the MoDeS^3 (Model-based Demonstrator for Smart and Safe Systems) [12] project.[7] The project incorporates a model railway consisting of track elements, such as *sections* and *turnouts*, and several *trains* moving on the track. The states of the turnouts (straight or divergent) and the motion of the trains (forward/backward and speed) are controlled externally by users. A major challenge of the project was to design a distributed controller that implements a safety logic to prevent the collision of trains by ensuring that there is at most one train on every section at any moment.

The distributed controller was designed as the composition of components belonging to the sections and turnouts. Each component receives signals from the environment related to the corresponding track element (e.g., about the presence and absence of trains), communicates with neighboring track elements to implement the safety logic and sends signals to the trains to proceed or stop. The components were designed using statecharts in a third party tool, Yakindu, and the whole system was integrated in the Gamma Composition Language, using different semantics at the different levels in the component hierarchy. At the lower level, components executed on the same physical controller were composed synchronously as a single piece of software. At the higher level, these composite components (that now supervise a collection of track elements) were composed in an asynchronous way to conform to the network-based communication between the physical controllers. There is a fundamental difference between these compositions, as component of a single software, or even components running on the same hardware have much higher guarantees than components relying on remote calls or messages over some network. This distinction will be handled by the different composition semantics introduced in the paper.

[7] Project repository with link to webpage: https://github.com/FTSRG/BME-MODES3/.

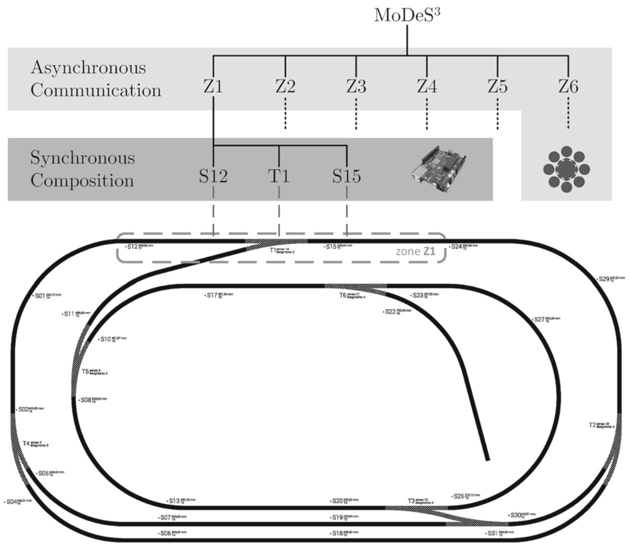


Fig. 1 The layout of the MoDeS^3 track. Note that segments are composed into zones, each supervised by a single embedded computer, which communicate through LAN

Figure 1 depicts the MoDeS^3 track, which consists of 6 turnouts and 25 sections. The distributed controller is deployed on 6 microcontrollers, each responsible for a single turnout and multiple section components. The turnout and the set of sections supervised by a particular microcontroller is called a *zone*.

2.2 Composite reactive modeling

The component-based logical design of reactive systems needs precise composition semantics that can be defined by means of a *model of computation*. We introduce the *asynchronous-reactive* model of computation for asynchronous systems (Sect. 2.2.1) as well as the *synchronous-reactive* and *cascade* models of computation for synchronous systems (Sect. 2.2.2) supported by our Gamma framework.

2.2.1 Asynchronous systems

In asynchronous systems [13], components represent concurrent entities that communicate with each other using message queues. Writing to a message queue succeeds instantly, whereas reading from an empty queue blocks the reader (nonblocking-write, blocking-read approach). Message delivery is assumed to be reliable; therefore, the sender does not receive nor expect any confirmation (send and forget approach). Messages arrive in the target message queue

in the same order they were sent.[8] A read operation always retrieves a single message from the queue. As extension, prioritized queues can be introduced to reorder the incoming messages and prefer the urgent ones in the read operation.
Asynchronous-reactive The asynchronous-reactive model describes concurrent components that are running continuously and react to events independently (therefore, execution frequency of system components is nondeterministic, but may be restricted with timing constraints). In the example, distributed controllers are most naturally modeled with this semantics, as different hardware and network topologies will most probably cause a different execution/delivery time in every case. In this context, synchronization of the components has to be ensured by communication protocols.

2.2.2 Synchronous systems

The synchronous domain has a notion of logical time and follows the semantics of synchronous programming languages [14–16]. In such systems, components communicate with each other using signals, which are transmitted and received through ports. The execution of components is driven by a *clock* that emits *ticks*. System components are executed in response to these ticks, and the execution of all components in a system is called a *cycle*. When a component is executed, it samples the signals from its incoming ports and transmits signals through its outgoing ports. Generally, components can be considered as functions mapping values from their incoming ports to their outgoing ports depending on their current state. The output signals of components are sustained until the next tick. As the input signals are sampled only at the beginning of execution, changes of signals during the executions are ignored. Contrary to the asynchronous domain, the components are also able to react to the *absence* of signals and even to a combination of signals.
Synchronous-reactive The synchronous-reactive model describes components that are executed in a lock step fashion upon every tick, that is, they all sample their input signals at the beginning of the cycle and process them concurrently. Thus, communication between components during a cycle is not possible, receiver components can process transmitted signals only in the next cycle (initiated by the subsequent tick). In the example, logical controllers of track segments may communicate like this, as the program running them has sufficient control to implement an explicit scheduling. Other cases, such as implementations in hardware or as PLC programs also call for this variant.

[8] These guarantees shall be achieved by proper control over the network and the appropriate protocols, which are considered as middleware that is not taken into account in the semantics. If this is not the case, other (unreliable) channel models can be considered by explicitly modeling the channel as a component for verification purposes.

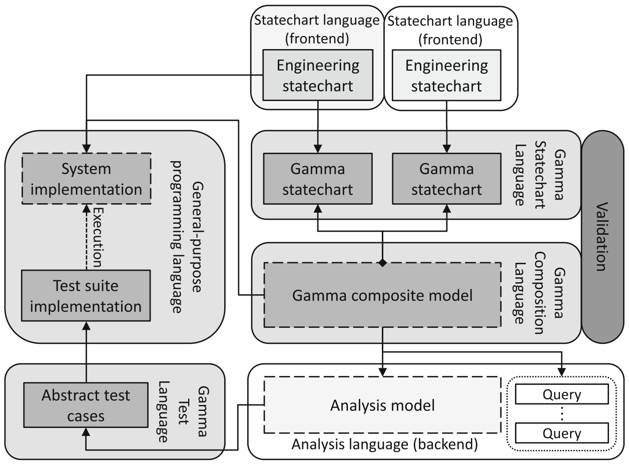

Statechart language (frontend)
Engineering statechart
Statechart language (frontend)
Engineering statechart
System implementation
Execution
Test suite implementation
General-purpose programming language
Gamma statechart
Gamma statechart
Gamma Statechart Language
Gamma composite model
Gamma Composition Language
Validation
Abstract test cases
Gamma Test Language
Analysis model
Query
Query
Analysis language (backend)

Fig. 2 The model transformation chains and languages of the Gamma framework. Rectangles represent models: solid border represents an atomic model, whereas dashed border represents a composite model. Dotted rectangles represent a set of models belonging together for fulfilling a more general purpose. Rectangles with moderately rounded corners represent languages. Rectangles with extensively rounded corners represent functionalities closely related to the usability of the language. Solid lines without a base symbol represent model transformations. Solid lines with a diamond symbol represent model composition. Dashed lines represent the ability of execution: the source model is capable of executing the target model

Cascade The cascade model supports the execution of components in a *linear way* (one after another in a specific order during a cycle) opposed to the concurrent, lock step execution of components of synchronous-reactive models. The execution of a cycle is also initiated by a tick; however, components sample their input signals right before they are executed and not at the beginning of the cycle. This enables communication between components during a cycle in a feed-forward way. Also, repeated execution of components during a single cycle is supported. In the example, a single logical controller may implement filters on its inputs and outputs, which does induce an execution order. Pipeline-like software or hardware implementations are best modeled with this variant.

2.3 The Gamma framework

The Gamma Statechart Composition Framework is an integrated toolset that supports model-driven design, validation, verification and code generation for component-based reactive systems. Figure 2 depicts the model transformation chains of the Gamma framework, the input and output models of these model transformations as well as the languages in which they can be defined, and the relations between these models.

There are two types of components in Gamma, atomic and composite components. The Gamma Statechart Language (GSL) supports the definition of atomic components in the form of statecharts. Also, this language enables the integra-

Table 1 Component types supported by the Gamma Composition Language

	Atomic	Composite
Synchronous	Statechart	Synchronous composite component, cascade composite component
Asynchronous	Asynchronous adapter	Asynchronous composite component

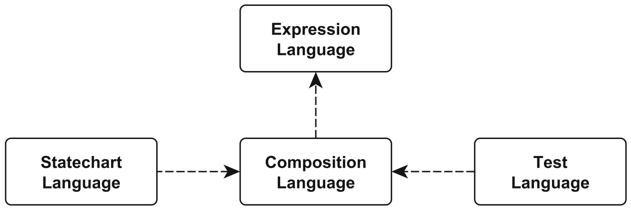


Fig. 3 Dependencies between the modeling languages of the Gamma framework

tion of component models (so-called engineering models) from off-the-shelf modeling tools, e.g., Yakindu or MagicDraw, as it is an intermediate formalism to which external models can be transformed. Composite components can be created by composing atomic components and other composite components based on well-defined interfaces using the Gamma Composition Language (GCL). Table 1 describes the component types supported by the GCL in terms of *synchronousness* and *compositeness*. The Gamma Test Language (GTL) supports the description of test cases as expected behaviors of Gamma components in response to input events, together with assertions regarding the state of the component. Expressions in these languages use the Gamma Expression Language (GEL). Figure 3 depicts the dependencies of these languages: GCL depends on GEL, and GTL and GSL depend on GCL.

To put these languages in context (and highlight the motivations for their design and formal semantics), below we summarize the automated functionalities provided by the Gamma framework.

Validation of models takes place at the level of atomic components as well as at system level by evaluating well-formedness constraints.

Once the entire system is modeled as a composition of statechart-based components, Gamma can *generate the implementation of the composition code* that wraps the existing (autogenerated) source code of atomic components. Accordingly, external code generators for statechart components can be integrated by implementing a plugin for the composition code generator that wraps the external code behind the interfaces generated by Gamma. Currently, Gamma supports the generation of Java code.

Formal verification of system models is provided by model checking [17], a technique that explores the behavior of the given model exhaustively against properties specified in mathematical logic (e.g., to check that unsafe states are not reachable). To assist software engineers, Gamma hides the inherent complexity of formal verification by (1) offering a pattern-based approach to specify the required properties [18], (2) using automated model transformations to the analysis models and queries of verification back-end tools like UPPAAL, and (3) back-annotating [19] the execution traces retrieved by the model checker (i.e., witnesses or counterexamples for the checked properties) to the system model in the form of abstract test cases. Gamma is also capable of generating a JUnit test case that replays the execution trace on the implementation.

We use the concepts of *model-based testing* [20] and apply the model checker to generate execution traces to achieve a designated test coverage based on automatically generated queries, namely that (1) each state of every statechart component is reached, or (2) each transition of every statechart is executed. These traces can be represented as abstract test cases in the GTL and also as concrete JUnit tests. With this test suite, designers can gain a certain level of confidence in the correctness of the Gamma toolchain (this way also validating the code generators) before deploying it into a critical environment.

3 Composition in the Gamma framework

This section introduces the composition structures in the Gamma framework. The design and formalization of the composition language can be considered as the most important conceptual result of this work, and thus, the most important novelty of Gamma 2.0 compared to the version introduced in [10]. After a brief introduction to the common structure of Gamma files, the following sections present the Gamma Composition Language (GCL) through the MoDeS^3 running example introduced in Sect. 2.1.

The compositional root elements of Gamma files are *packages*. A Gamma model may contain one or more packages. They can contain constant declarations, interface definitions (Sect. 3.1) and Gamma components (Sects. 3.2–3.5). Elements declared in a package can be reused in other packages by importing the containing package, using relative or absolute paths to the other file (with or without file extension).

An example package can be seen below, which imports another package to reuse the component *Z1* in the declaration of an asynchronous adapter component *Z1Adapter* (see Sect. 3.5) and defines a constant integer.

```
package z1_adapter
// Importing other packages
import "model/Components/Z1"
// Constants can be used in the definitions
const QUEUE_CAPACITY : integer := 16
// Component definition
async Z1Adapter of component z1 : Z1 [
    ...
] {...}
```

Well-formedness constraint there must not be any circular dependencies between the packages.

3.1 Interfaces and ports

Event Events represent discrete occurrences of some importance with optional parameters (the “payload”). Every event has a source and may trigger some behavior. In the framework, events may describe the sending/receiving of *signals* in the case of synchronous components or *messages* in the case of asynchronous components.
Interface The GCL supports the definition of interfaces, which serve as contracts between interacting components of Gamma models describing the types of events a component can *dispatch* (*out* or *inout* events) or *receive* (*in* or *inout* events). Furthermore, an interface may *inherit* all declared events from zero or more other interfaces to enable polymorphism (denoted by $A \prec B$ if $B = A$ or if B extends A or if B extends an interface C such that $A \prec C$). The fragment of the GCL metamodel describing interfaces and events is depicted in Fig. 4a. An example for inheritance can be seen below, where *Derived* contains two events: *foo* (which has an integer parameter) and *bar*.

```
interface Base {
  in event foo(param : integer)
}

interface Derived extends Base {
  inout event bar
}
```

The interfaces used in the safety logic model of MoDeS^3 are depicted in Fig. 5a. Interface *Protocol* contains events used for the communication between track elements (sections and turnouts) to implement the safety logic protocol. *SectionControl* contains events controlling traffic on a certain section (*enabled* or *disabled*), while *TurnoutControl* can be used by the environment to change the state of a certain turnout. The events of *Train* denote the arrival end departure of a train on a certain section. Finally, *TrainControl* events are also used by the environment to move the train forward or backward (the absence of these events mean the train has to be idle).

Port Ports serve as communication endpoints of component instances. Each event is dispatched or received through a port. Each port realizes a single[9] interface by defining an interface realization. Components do not realize interfaces directly.
Interface realization Interfaces can be realized in either *provided* or *required* mode. The difference is explained using the interface definition *Derived* from before.

– *Provided mode* ports dispatch and receive events according to the direction they have been declared. For example, the component owning the port will be able to dispatch *bar* (*out* or *inout* events) and receive *foo* and *bar* (*in* or *inout* events) through this port. Such a port would be defined as:

```
port ProviderPort : provides Derived
```

– *Required mode* interfaces are conjugated,[10] i.e., *in* events will be dispatched (because the receiver provides an *in* event), and *out* events will be received (the dispatcher provides an *out* event). A component owning the example port will be able to dispatch events *foo* and *bar* (declared *in* or *inout* events in the interface) and receive event *bar* (declared *out* or *inout* events) through this port:

```
port RequirerPort : requires Derived
```

If two ports realize interfaces A in required mode and B in provided mode such that $A \prec B$, the two ports are *compatible* (i.e., they can be connected) since every event that A requires is present on B in the opposite direction. This is the semantic difference between required and provided realizations—events on a required interface must be present on the other side of the connection, while provided interfaces may have unconnected events.

We say that a port is a *broadcast* port if (1) the interface realization mode is ‘provided’ and (2) the realized interface contains only *out* events *OR* (1) the interface realization mode is ‘required’ and (2) the realized interface contains only *in* events. Unlike other ports, a broadcast port can be connected to multiple ports without the need to handle multiple occurrences of the same event, because every *in* event has a single source only (see the semantics and well-formedness constraints of channels in Sect. 3.3).

Even though Gamma interfaces may contain both *in* and *out* events, the reason we use only *out* events in the MoDeS^3 safety logic is to have broadcast ports. The directed events, provided-required modes and broadcast ports, offer a flexible way to model communication interfaces. Our goal with this design is to explore the possibilities of interface-based communication in reactive systems while giving freedom to

[9] The single interface may inherit from many others, so this is not a limitation in practice.

[10] This is equivalent to interface conjugation in SysML.

users to employ their own convention (such as input/output interfaces).

3.2 Components

Components serve as *types* of component instances. They can have parameters that can be used in their bodies just like constant declarations. Components may be atomic or composite and synchronous or asynchronous (see Sects. 3.4–3.5). A component can have zero or more ports. In Gamma, any interaction of a component, whether originating from an internal subcomponent or the implementation of an atomic component, must occur through a port.[11] This ensures that external dependencies and interactions are explicitly modeled, leading to fully encapsulated units of behavior that can be logically analyzed in and of itself,[12] facilitating contract-based and compositional analysis and monitoring. The fragment of the GCL metamodel describing component types is depicted in Fig. 4c.

Atomic Component An atomic Gamma component is an abstract element with the following properties:

- a set of states with a well-defined initial state,
- a set of input and output events collected from ports of the component,
- a deterministic[13] transition function describing the behavior of the component, i.e., the new state and raised output events in response to input events.

The built-in Gamma Statechart Language (GSL) supports the definition of atomic components with statecharts. It offers elements similar to that of UML and SysML, such as composite states and orthogonal regions for describing concurrent execution, history and variables for expressing memory, timeouts for modeling timed behavior, and choices for complex transitions. Some elements that are not supported include completion-related constructs and internal transitions (the latter can be mapped to subregions though). Additional constructs not present in UML/SysML statecharts are complex triggers (reacting to a combination of events) and metadata annotations that can refine the execution semantics (e.g., defining the means of conflict resolution when multiple transitions are enabled). Most of the differences originate from the fact that UML/SysML statecharts have an essentially asynchronous semantics, while the GSL was designed to support synchronous-reactive statecharts (see Sect. 4 for a formal discussion of semantics).

In the current implementation, the GSL provides a common basis for all functionalities of the Gamma framework. However, the definition of atomic Gamma components is not restricted to the GSL, the framework can be extended with other languages as long as it defines the behavior in terms of the properties above.

Component instance Component instances are individual reactive elements with their own internal state and a (constant) value assigned to each of their parameters. Each instance has a type (a component definition) that determines the ports on which the component instance can communicate as well as the internal states it can assume and the transitions it can fire. Code generation and analysis always happen on an instance with all parameters bound to a constant value. An example component instance *S12* can be seen below. Note that whether the component instance is synchronous or asynchronous is defined by its type (see Sects. 3.4, 3.5).

```
component S12 : Section
```

Well-formedness constraint each parameter of a component type must be bound to a value with an appropriate type in every instance of the component.

Composite component Composite components are defined in terms of one or more *component instances* (either atomic or composite, therefore supporting hierarchical composition). The behavior of the composite component is defined by the interaction of its *constituent components*. Sects. 3.4 and 3.5 present the supported composite component types in detail, after Sect. 3.3 introduced common communication elements (supported in each composite component type) required to define interactions between constituent components.

3.3 Communication elements

The fragment of the GCL metamodel describing communication elements is depicted in Fig. 4b.

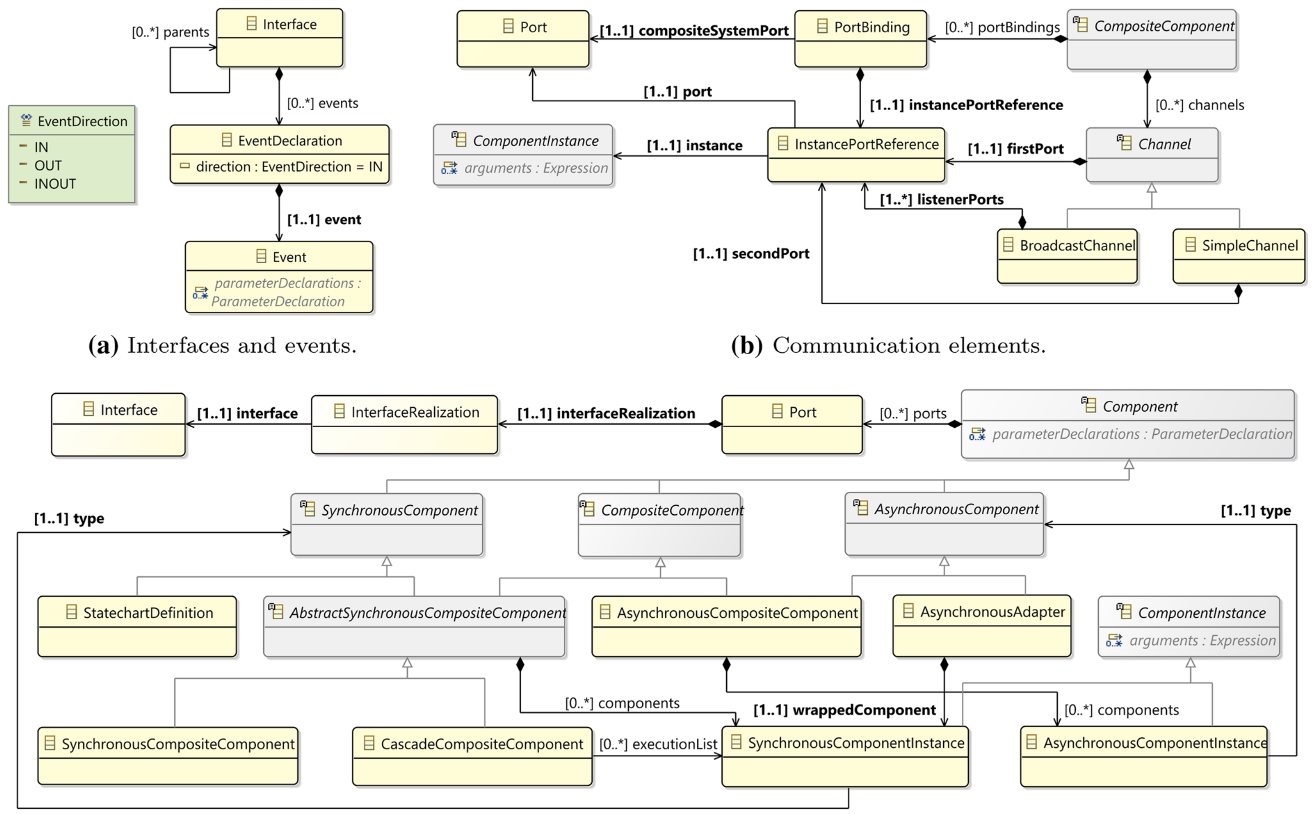
Instance port reference The instance port reference element is the qualified name of a port of a component instance; thus, it refers to a single port and a single component instance. An example instance port reference can be seen below.

```
S12.ProtocolOutCCW
```

[11] Note that this is only a syntactical requirement, as direct interaction between subcomponents can always be transformed to strict hierarchical communication by adding extra ports and routing events up and down in the hierarchy. A component with no ports cannot interact with its environment, but may have internal (e.g., timed) behavior that can be analyzed.

[12] Like most similar formalisms, Gamma does not consider the operational environment or the executing (virtual) machine directly. Interactions resulting from such unconsidered circumstances can very well influence the behavior of a component in many specific ways. While we do not consider these directly, the effect of the environment and different fault and interaction models can be captured by adding additional ports to the components and substituting simple communication channels with one or more components representing the communication platform.

[13] This definition is extensible to nondeterministic components, but this would be relevant only in the case of environmental models, which is not in the primary scope of the current work.



(a) Interfaces and events. **(b)** Communication elements.

(c) Component types.

Fig. 4 Fragments of the GCL metamodel related to composite components

Well-formedness constraint the port must be defined on the type of the component instance.

Port binding The port binding element is responsible for mapping the ports of a composite component ("system ports"), to the ports of constituent component instances.[14] Therefore, it refers to a single system port and a single instance port reference. All events received on the particular system port will be transmitted to the port of the associated component instance, and all the events dispatched through the particular instance port will be transmitted to the system port. An example port binding can be seen below.

```
bind S12ProtocolOutCCW -> S12.ProtocolOutCCW
```

Well-formedness constraints (1) in the case of synchronous and cascade composite components (see Sect. 3.4), a non-broadcast system port must be mapped to a single port of a constituent component instance; (2) in the case of synchronous and cascade composite components, a non-broadcast port of a constituent component instance can be connected to at most one system port; (3) both ports must realize their interface in the same mode; (4) given that the system port realizes interface S and the port of the component instance realizes interface C, if the realization mode is *required* then $C \prec S$ (S extends C) and if it is *provided* then $S \prec C$.

Channel Channels are responsible for the connection of component instance ports. Technically, they use instance port references to refer to the endpoints. There are two types of channels: *simple* channels and *broadcast* channels.

– *Simple channels* support the connection of two ports. An example of a simple channel can be seen below (connecting provided port *ProtocolOutCW* of *Section* instance *S12* and required port *ProtocolInStraight* of *Turnout* instance *T1* in composite component *Z1* as defined in Fig. 5b).

```
channel [S12.ProtocolOutCW] -o)-
    [T1.ProtocolInStraight]
```

Well-formedness constraints (1) the connected ports must be *compatible* (as defined in Sect. 3.1); (2) in the case of synchronous and cascade composite components (see Sect. 3.4) a non-broadcast port must not be referred to in more than one channel or port binding (to avoid race conditions, since synchronous components cannot han-

[14] The approach is again similar to using proxy ports and binding connectors in SysML.

```
interface Protocol {
  out event occupied
  out event unoccupied
  out event go
  out event stop
}

interface SectionControl {
  out event enable
  out event disable
}

interface TurnoutControl {
  out event turnoutStraight
  out event turnoutDivergent
}

interface Train {
  out event occupy
  out event unoccupy
}

interface TrainControl {
  out event moveForward
  out event moveBackward
}
```

(a) Interfaces.

```
package z1
import "model/Interface/Interface"
import "model/Section/Section"
import "model/Turnout/Turnout"
sync Z1 [
  // Ports on the borders of the zone
  port S12ProtocolInCCW : requires Protocol
  port S12ProtocolOutCCW : provides Protocol
  port T1ProtocolInDivergent : requires Protocol
  port T1ProtocolOutDivergent : provides Protocol
  port S15ProtocolInCW : requires Protocol
  port S15ProtocolOutCW : provides Protocol
  // Control ports for track elements
  port S12Control : provides SectionControl
  port T1Turnout : requires TurnoutControl
  port S15Control : provides SectionControl
  // Train ports for track elements
  port S12Train : requires Train
  port T1Train : requires Train
  port S15Train : requires Train
] {
  // Section and turnout components
  component S15 : SectionStatechart
  component T1 : TurnoutStatechart
  component S12 : SectionStatechart
  // Binding system ports to ports of components
  bind S12ProtocolInCCW -> S12.ProtocolInCCW
  bind S12ProtocolOutCCW -> S12.ProtocolOutCCW
  ...
  // Binding control ports and train ports
  bind S12Control -> S12.SectionControl
  bind S12Train -> S12.Train
  ...
  // Connecting elements of the zone
  channel [S15.ProtocolOutCCW] -o)-
    [T1.ProtocolInTop]
  channel [T1.ProtocolOutTop] -o)-
    [S15.ProtocolInCCW]
  channel [S12.ProtocolOutCW] -o)-
    [T1.ProtocolInStraight]
  channel [T1.ProtocolOutStraight] -o)-
    [S12.ProtocolInCW]
}
```

(b) The definition of Zone No. 1.

```
package z1
...
cascade Z1 [
  // Ports on the borders of the zone
  port T1ProtocolInDivergent :
    requires Protocol
  port T1ProtocolOutDivergent :
    provides Protocol
  ...
] {
  // Section and turnout components
  component S15 : SectionStatechart
  component T1 : TurnoutStatechart
  component S12 : SectionStatechart
  // Execution order of component instances
  execute T1, S15, S12, T1, S15, S12
  // Binding system ports
  ...
  // Connecting elements of the zone
  ...
}
```

(c) The cascade composite component variation of Zone No. 1, where each component instance is executed twice in an execution cycle.

```
package z1_adapter
import "model/Z1/Z1"
const QUEUE_CAPACITY : integer := 16
adapter Z1Adapter of component z1 : Z1 [
  // No additional ports on the adapter
] {
  // Run wrapped component on any kind of received event
  when any / run
  // Protocol messages go in a higher priority queue
  queue protocolMessages (priority = 2,
      capacity = QUEUE_CAPACITY) {
    T1ProtocolInDivergent.any,
    T1ProtocolOutDivergent.any, S15ProtocolInCW.any,
    S15ProtocolOutCW.any, S12ProtocolInCCW.any,
    S12ProtocolOutCCW.any
  }
  // Other control messages go in a lower priority queue
  queue controlMessages (priority = 1,
      capacity = QUEUE_CAPACITY) {
    S15Restart.any, S15Control.any, T1Turnout.any,
    S12Restart.any, S12Control.any, S15Train.any,
    T1Train.any, S12Train.any
  }
}
```

(d) The UML-like asynchronous adaptation of Zone No. 1.

```
package z1_adapter
import "model/Z1/Z1"
adapter Z1Adapter of component z1 : Z1 [
  // No additional ports on the adapter
] {
  // Clock emitting a tick every millisecond
  clock millisecondClock(rate = 1 ms)
  // Run the component on clock events
  when millisecondClock / run
  // Storing clock messages in a higher priority queue
  queue clockMessages (priority = 0, capacity = 8) {
    millisecondClock
  }
  // Storing other messages in lower priority queues
  ...
}
```

(e) The timed asynchronous adaptation of Zone No. 1.

Fig. 5 Model snippets related to the MoDeS^3 safety logic

dle multiple occurrences of the same event at the same time).

– *Broadcast channels* support sending events to multiple target ports. Such channels refer to (1) a single broadcast port and (2) one or more other ports. In this case, the direction of event transmission is determined: the broadcast port dispatches events and all the other ports connected to it receive them. A broadcast channel example can be seen below (connecting broadcast port *Train* of component instance *train* to the required ports *Train* of component instances *lowerLevelModel* and *higherLevelModel*).

```
channel [train.Train] —o)—
    [lowerLevelModel.Train, higherLevelModel.Train]
```

Well-formedness constraints (1) the connected ports must be *compatible*; (2) non-broadcast ports must not be referred to in more than one channel or port binding in the case of synchronous and cascade composite components (but broadcast ports may).

3.4 Synchronous components

This section introduces the synchronous components of the GCL, including their informal behavioral semantics. Synchronous components can be either atomic components (e.g., *statechart definitions*, introduced in Sect. 3.2) or composite components (*synchronous* and *cascade*). See the left side of Fig. 4c for the relevant part of the metamodel.

Synchronous components represent systems that communicate in a synchronous manner using *signals*. Their execution is scheduled by a scheduler, which can be an asynchronous adapter component (see Sect. 3.5) or a custom scheduler implementation invoking a component from time to time. The execution of a synchronous component conforms to a turn-based semantics. A turn is called a *cycle*. In a cycle, synchronous components process incoming signals and produce output signals in accordance with their internal states. Output signals are present for a single execution cycle only, i.e., the signal disappears after one cycle (but another output signal may be produced then). The semantics of synchronous components was designed in accordance with the synchronous-reactive and cascade models of computation presented in Sect. 2.2.2.

Synchronous composite component The execution of a synchronous composite component is based on the execution of its contained component instances. When a single component instance is executed, it (1) processes signals received in the last execution cycle, (2) assumes a new state according to the processed signals and (3) produces signals that can be received *in the next cycle* by others or itself.

In each cycle, all component instances of the particular composite component are executed. As contained components *cannot affect* each other during a single execution cycle, the execution order of contained components does not matter. The results of an execution cycle are the same regardless of the execution order of the components, which is a key feature of synchronous composite components.

In Fig. 5b, synchronous composite component *Z1* is depicted. Note how the previously introduced endpoint and communication elements can be used to establish a strongly coupled composite component.

Cascade composite component Cascade composite components are structurally similar to synchronous composite components, but their execution semantics differ. The execution of a cascade composite component also consists of cycles. In a single cycle, all constituent components are executed in a specific order, which can be based on either an *execution list* defining the order of the execution of components (has to be defined explicitly), or the declaration order of the component instances (default). The execution list can contain a particular component instance one or more times, i.e., a component instance can be executed multiple times in a single cycle. If no execution list is defined, each component instance is executed exactly once in the order of declaration.

When a component is executed, it (1) processes all incoming signals, (2) assumes a new state according to the processed signals and (3) produces output signals. However, the effect of a signal is observable *immediately* in the same execution cycle to other component instances executed later (feed-forward connections), and not only in the next one as in synchronous composite components. Accordingly, signals sent through feedback connections, i.e., when a component instance sends a signal to another one that comes earlier in the execution order, are observable in the *next* execution cycle. Note that cascade and synchronous composite components are semantically incompatible, that is, there are models in both variants that cannot be simulated by a model in the other due to the differences in the signal transfer (with respect to trace equivalence).

Well-formedness constraints if an execution list is defined, it must contain each defined component instance at least once.

Figure 5c depicts a variant of synchronous composite component *Z1* by defining an execution list. Note that the definition of cascade and synchronous composite components differ in the *cascade* keyword at the beginning of the component definition and the optional definition of an execution list.

3.5 Asynchronous components

This section introduces the asynchronous components of the GCL. Asynchronous components can be either atomic components (*asynchronous adapter*) or composite components (*asynchronous composite component*).

Asynchronous components represent independently running component instances. There is no guarantee on the exact running time or the running frequency of such components, so different interleavings of component executions can occur nondeterministically. Therefore, asynchronous components may receive multiple messages between two executions, which requires communication with *buffered messages*. The semantics of asynchronous components was designed in accordance with the asynchronous-reactive model of computation presented in Sect. 2.2.1.

Asynchronous adapter An asynchronous adapter wraps a single synchronous component instance, turning it into an asynchronous component. The adapter can be regarded as a composite component with a single synchronous instance, such that adapters implicitly have all ports of the wrapped synchronous component—and optionally some additional ones for the reception of control messages (see below).

Furthermore, an asynchronous adapter has one or more **message queues**, which store the incoming messages of the component and have multiple attributes:

– *Capacity* specifies the maximum number of messages that can be stored in the particular queue. If a queue is full and an additional message is received, the message is discarded. Note that the absence of this attribute implies a queue with an unlimited size, which may lead to an infinite state space. Therefore, the capacity attribute has a great impact on verification and may also be necessary for code generation.
– *Priority* of a queue specifies the order in which the contents of message queues are retrieved during the execution of the asynchronous component. A message is always retrieved from a non-empty queue with the highest priority. Priority values can be non-negative integers, where higher values represent higher priorities.
– *Event references* specify the types of messages that can be stored in the particular message queue (demultiplexing incoming messages into message queues). The language allows referring to a single event of a single port, all events of a port, or all events of the component. If a particular message could be stored in multiple message queues, the one declared first will be used, supporting hierarchical filters.

During execution, messages are retrieved from messages queues *one by one*. A message is always taken from the highest priority non-empty queue. If the particular message was received on a port that is implicitly derived from the wrapped component, the message is converted to a signal (as synchronous components communicate with signals) and transmitted to the wrapped synchronous component (potentially overwriting previously sent signals). If it was received on a port explicitly defined on the adapter component, the message is not transmitted.

An asynchronous adapter also has one or more **control specifications**, which specify the messages that are able to trigger the execution of the wrapped component. If a specified message arrives, the wrapped component is executed. Note that signals derived from messages are transmitted to the wrapped component before execution, so a triggered execution will process the signal even if the control specifications is specified for the corresponding message.

Finally, asynchronous adapters can contain zero or more **clocks**, which emit tick events at defined timed intervals. Such time intervals can be defined with the *rate* attribute. Currently, seconds and milliseconds are supported as units of time. Tick events also have to be assigned to a queue and can be handled in control specifications similarly to regular events received from ports.

To demonstrate the flexibility of this control specification-based approach, we present two different execution semantics, reusing the *Z1* synchronous composite component model.

– In Fig. 5d, a single control specification is defined that triggers on the "*any* event." In this case, every time an event is retrieved from a message queue, the wrapped component gets executed. This behavior is similar to the semantics of UML statecharts [21].
– In Fig. 5e, a single control specification is defined that triggers on the *ticks of a clock*. In this case, the wrapped component gets executed in defined periods of time and processes the events that arrive between the ticks.

Well-formedness constraints (1) the rate attribute of clocks and the priority and capacity attributes of message queues must have non-negative integer values; (2) events specified in control specifications must be input events; (3) event references in message queues must refer to input events; (4) each input event of the asynchronous adapter must be referred in a single message queue.

Asynchronous composite component Asynchronous composite components support the hierarchical definition of asynchronous components. Similarly to synchronous composite components, an asynchronous composite component consists of port bindings and channels in addition to *asynchronous* component instances. Synchronous components must be wrapped in an asynchronous adapter to include them in asynchronous composite components.

In Fig. 6, the entire MoDeS^3 safety logic is defined as an asynchronous composite component, which contains the wrapped synchronous zone models.

```
package modes_track
import "model/Interface/Interface"
import "model/Z1/Z1Adapter"
...
import "model/Z6/Z6Adapter"
async ModesTrack [
  // Turnout control ports
  port T1Turnout : requires TurnoutControl
  ...
  port T6Turnout : requires TurnoutControl
  // Train ports
  port T1Train : requires Train
  ...
  port S31Train : requires Train
  // Section control ports
  port S01Control : provides SectionControl
  ...
  port S31Control: provides SectionControl
] {
  // Instances of the wrapped zone models
  component Z1 : Z1Adapter
  ...
  component Z6 : Z6Adapter
  // Binding system ports
  bind T1Turnout -> Z1.T1Turnout
  ...
  bind T6Turnout -> Z6.T6Turnout
  bind T1Train -> Z1.T1Train
  ...
  bind S31Train -> Z2.S31Train
  bind S01Control -> Z4.S01Control
  ...
  bind S31Control -> Z2.S31Control
  // Connecting zones
  // Z1
  channel [Z1.T1ProtocolOutDivergent] -o)-
    [Z5.S11ProtocolInCW]
  channel [Z5.S11ProtocolOutCW] -o)-
    [Z1.T1ProtocolInDivergent]
  ...
  // Z2
  channel [Z2.T2ProtocolOutDivergent] -o)-
    [Z3.S30ProtocolInCCW]
  channel [Z3.S30ProtocolOutCCW] -o)-
    [Z2.T2ProtocolInDivergent]
  ...
  // Z3
  channel [Z3.S19ProtocolOutCW] -o)-
    [Z4.S07ProtocolInCCW]
  channel [Z4.S07ProtocolOutCCW] -o)-
    [Z3.S19ProtocolInCW]
  ...
  // Z5
  channel [Z5.S10ProtocolOutCW ] -o)-
    [Z6.S17ProtocolInCCW]
  channel [Z6.S17ProtocolOutCCW ] -o)-
    [Z5.S10ProtocolInCW]
}
```

Fig. 6 The high-level MoDeS^3 safety logic model

4 Formal semantics of the Gamma Composition Language

This section presents the formal structures and semantics of the GCL, which distinguishes it from other informal and semi-formal modeling languages. Subsections include short discussions about additional practical and theoretical aspects, design decisions and consequences on verification and code generation.

The section starts with the definition of events and related structures (Sect. 4.1), and then event vectors related to signals (Sect. 4.2) and the syntactic definition of a synchronous component (Sect. 4.3) are introduced. Next, synchronous composite components and cascade composite components are formalized both syntactically and semantically (Sects. 4.4, 4.5). After defining event sequences (Sect. 4.6) and asynchronous components (Sect. 4.7), asynchronous adapters and their semantics are presented (Sect. 4.8). The section concludes with the definition of asynchronous composite components (Sect. 4.9) and their semantics in terms of their environment (Sect. 4.10) as well as messages, occurrences and execution traces (Sect. 4.11). To help the reader find their way through the following pages, Appendix lists the symbols used in the definitions along with a short description.

4.1 Events

Definitions of Sect. 4 consider individual events only, since ports and interfaces are syntactic sugar that facilitate the structuring of syntactic contracts. 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 or the change of situation (state). Given a set of events E, the finite domains of event parameters are defined by the domain function $\mathcal{D} : E \rightarrow 2^{\{d_1,\ldots,d_n\}}$. 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).

Discussion An event represents a declaration only. Furthermore, an event instance is not an event occurrence, as there may be many occurrences of a single event instance.

4.2 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.

Discussion Event vectors do need memory to represent them at runtime. Event vectors can be regarded as "nullable" variables dedicated to each event holding the occurrence and the parameters of that particular event (if any). The number of events and the size of their parameter domains therefore directly affect the size of the state vector in formal verification or the memory requirements of an implementation.

4.3 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.
– $T : S \times V_I \to S \times V_O$ 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.[15]

If $\ominus$ is a timed component, we allow S to track the values of *clock variables* [22], assuming that whenever a clock variable in *any component of the whole system* is increased by (a positive) Δt, *all other clock variables* in *any component of the whole system* are also increased by the same Δt. Also, elapsing time *may not* trigger an internal execution of the component, i.e., the state of the component apart from the values of the clock variables may only change in accordance with T. Finally, we require the execution of transitions to be atomic and instantaneous, i.e., time may not elapse during the execution of a single transition (regardless of whether the component is atomic or composite).

Discussion Synchronous components take an event vector as an input and generate an event vector as an output. Considering that the synchronous component is a statechart, the definition is closest to the virtual finite state machine formalism introduced in [23]. We chose this formalism because it harmonizes with the synchronous-reactive domain—as components are executed in a lock-step fashion, there may be a need to react to multiple events or a combination of events at the same time, which can be handled by complex triggers (included in the GSL). Furthermore, as events can occur *at the same time*, we do not have to analyze unnecessary interleavings introduced by the often arbitrary order of sequential message passing. Nevertheless, the more widespread event-driven finite state machine, which is the basis of most commonly used statechart formalisms, is also suitable to describe a component. However, those components may be triggered by event vectors with a single "non-null" event instance only, and we must ensure that they are executed *every time* a signal arrives (see the asynchronous adapter in Sect. 4.8).

Note that this definition describes an abstract behavioral contract that can be implemented by multiple formalisms. This is the way the framework supports the integration of external tools and formalisms. The rest of the section is about the semantics of scheduling components, which is the main focus of our work.

4.4 Synchronous composite component

Recall that a synchronous composite component is defined by instantiating a set of constituent components, exporting input and output ports (events in the formal case) by port bindings and defining channels (connecting events instead of ports in the formal case).

Definition 4 A synchronous composite component is a tuple $\textcircled{s} = (\mathbf{C}, I, O, \rightleftharpoons)$:

– $\mathbf{C} = \{\ominus_1, \ldots, \ominus_K\}$ is the set of synchronous components constituting the composite component, each component being $\ominus_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
– $I \subseteq \hat{I}$ is the set of exported input events, where $\hat{I} = \bigcup_{k=1}^{K} I_k$.
– $O \subseteq \hat{O}$ is the set of exported output events, where $\hat{O} = \bigcup_{k=1}^{K} O_k$.
– $\rightleftharpoons : \hat{I} \setminus I \to \hat{O}$ is the *channel* function that assigns an output as the source of events to every input, with the restriction that it must not be defined for elements of I, that is, an input is either linked to an output or is an exported input. We demand that for each $e \in \hat{I} \setminus I$, $\mathcal{D}(e) = \mathcal{D}(\rightleftharpoons(e))$.

Discussion Note that *port binding* elements are not present in the definition, instead *exported* events are defined. This implies that events bound together in a composite model are handled as if they were the same, they are not differentiated in any way, as they represent the same occurrence.

Semantics To understand the semantics of synchronous composite components, i.e., its behavior as a synchronous component, recall that output signals produced by a component are sampled by other components in the next execution

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

cycle only. To describe this behavior, we extend the combined state space of the constituent components with the last output event vector *of all constituent components*. An execution cycle is described by the emergent transition relation of the composite component.

Definition 5 A synchronous composite component ⓢ is itself a synchronous component $ⓢ\rangle\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- $S = S_1 \times \ldots \times S_K \times V_{\hat{O}}$ is the set of potential states, derived as all possible combinations of the potential states of the constituent components and the last output event vector of every component.
- $s^0 = (s_1^0, \ldots, s_K^0, \perp_{\hat{O}})$ is the initial state, where every constituent component is in its initial state and the last output event vector $\perp_{\hat{O}} \in V_{\hat{O}}$ assigns $\perp$ to every output event ($\forall e \in \hat{O} : \perp_{\hat{O}}(e) = \perp$).
- I is the set of exported input events and O is the set of exported output events as defined in Definition 4 (remember that we denote $I \cup O$ by E).
- $\mathcal{D}$ is implicitly defined by $\mathcal{D}_k$, as $\hat{\mathcal{D}} = \bigcup_{k=1}^{K} \mathcal{D}_k$ and $\mathcal{D}(e) = \hat{\mathcal{D}}(e)$ for all $e \in E$.
- The transition function is defined as $T\big((s_1, \ldots, s_K, v_{\hat{O}}), v_I\big) = \big((s_1', \ldots, s_K', v_{\hat{O}}'), v_O\big)$, where:
  - For each input event $e \in \hat{I}$ of any constituent component, let $v_{\hat{I}}(e) = v_I(e)$ if $e \in I$ or $v_{\hat{I}}(e) = v_{\hat{O}}(\rightleftharpoons(e))$ otherwise. Note that $v_{\hat{I}}$ implicitly defines every v_{I_k} as well, because $v_{\hat{I}} = \bigcup_{k=1}^{K} v_{I_k}$.
  - The next state s_k' of every component corresponds to the transition function T_k such that $T_k(s_k, v_{I_k}) = (s_k', v_{O_k}')$.
  - $v_{\hat{O}}' = \bigcup_{k=1}^{K} v_{O_k}'$ is the new vector of last output events.
  - The output of the composite component for each exported output $e \in O$ is defined by the output of the constituent components: $v_O(e) = v_{\hat{O}}'(e)$.

Discussion When executing a synchronous composite component, its constituent components either react to an external input (in the case of exported inputs) or to the output of a constituent component (including themselves) *from the previous execution cycle.* This prevents any interaction between the components during a single execution cycle, allowing to execute the components in an *arbitrary order*, essentially performing *partial order reduction* statically. This key feature greatly reduces the size of the state space, making the synchronous-reactive domain suitable for formal verification. Additionally, the definition enables to connect a single output to multiple inputs of components; however, an input can be connected only to a single output.

4.5 Cascade composite component

The syntactic definition of cascade composite components is the same as that of synchronous composite components, apart from the additional definition of the execution order of constituent components.

Definition 6 A cascade composite component is a tuple $ⓒ = (\mathbf{C}, X, I, O, \rightleftharpoons)$:

- $\mathbf{C} = \{\ominus_1, \ldots, \ominus_K\}$ is the set of synchronous components constituting the composite component, each component being $\ominus_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
- $X \in \mathbf{C}^*$ is a finite *ordered sequence* (with potential repetitions) of synchronous components called the *execution sequence* specifying the components to be executed in an execution cycle.
- $I \subseteq \hat{I}$ is the set of exported input events, where $\hat{I} = \bigcup_{k=1}^{K} I_k$.
- $O \subseteq \hat{O}$ is the set of exported output events, where $\hat{O} = \bigcup_{k=1}^{K} O_k$.
- $\rightleftharpoons : \hat{I} \setminus I \to \hat{O}$ is the *channel* function that assigns an output as the source of events to every input, with the restriction that it must not be defined for elements of I, that is, an input is either linked to an output or is an exported input. We demand that for each $e \in \hat{I} \setminus I$, $\mathcal{D}(e) = \mathcal{D}(\rightleftharpoons(e))$.

Semantics Cascade composite components do not delay the internal signals between constituent components executed after each other (feed-forward signals); therefore, the effect of an event is computed in a single run. Signals sent to components that are not executed anymore in the current execution cycle (feedback signals) are saved for the next cycle, just like in synchronous composite components.

Definition 7 A cascade composite component ⓒ is itself a synchronous component $ⓒ\rangle\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- $S = S_1 \times \ldots \times S_K \times V_{\hat{O}}$ is the set of potential states, derived as all possible combinations of the potential states of the constituent components and the last output event vector of every component.
- $s^0 = (s_1^0, \ldots, s_K^0, \perp_{\hat{O}})$ is the initial state, where every constituent component is in its initial state and the last output event vector $\perp_{\hat{O}} \in V_{\hat{O}}$ assigns $\perp$ to every output event ($\forall e \in \hat{O} : \perp_{\hat{O}}(e) = \perp$).
- I is the set of exported input events and O is the set of exported output events as defined in Definition 6 (recall that $I \cup O$ is denoted by E).
- $\mathcal{D}$ is implicitly defined by $\mathcal{D}_k$, as $\hat{\mathcal{D}} = \bigcup_{k=1}^{K} \mathcal{D}_k$ and $\mathcal{D}(e) = \hat{\mathcal{D}}(e)$ for all $e \in E$.

– The transition function is $T\big((s_1,\ldots,s_K,v_{\hat{O}}),v_I\big) = \big((s'_1,\ldots,s'_K,v'_{\hat{O}}),v_O\big)$, computed iteratively for every $X[i]$ $(1 \le i \le n, n = |X|)$:
  – Let $(s_1^{(0)},\ldots,s_K^{(0)},v_{\hat{O}}^{(0)}) = (s_1,\ldots,s_K,v_{\hat{O}})$ (the source state).
  – Assume that $X[i] = \ominus_k$. To obtain $(s_1^{(i)},\ldots,s_K^{(i)},v_{\hat{O}}^{(i)})$, we apply $T_k(s_k^{(i-1)},v_{I_k}) = (s_k^{(i)},v_{O_k})$ to compute $s_k^{(i)}$ and v_{O_k}, where for all $e \in I_k$,

$$v_{I_k}(e) = \begin{cases} v_I(e) & \text{if } e \in I \text{ and this is} \\ & \text{the first execution of } \ominus_k, \\ \bot & \text{if } e \in I \text{ and it is not} \\ & \text{the first execution, and} \\ v_{\hat{O}}^{(i-1)}(\rightleftharpoons(e)) & \text{if } e \notin I. \end{cases}$$

  The state of other components $\ominus_j \in \mathbf{C}$ $(j \neq k)$ remains the same $(s_j^{(i)} = s_j^{(i-1)})$. The last output event vector is updated with v_{O_k}: for all $e \in \hat{O}$, $v_{\hat{O}}^{(i)}(e) = v_{O_i}(e)$ if $e \in O_k$, and $v_{\hat{O}}^{(i)}(e) = v_{\hat{O}}^{(i-1)}(e)$ otherwise.
  – Finally, $s'_k = s_k^{(n)}$ for every $\ominus_k \in \mathbf{C}$ and $v_O(e) = v_{\hat{O}}^{(n)}(e)$ for every $e \in O$.

Discussion The raison d'etre of the cascade composite semantic variant is twofold. First, even though it requires the same amount of memory to represent as synchronous composite components (see the definition of *S*), the effect of an input event on output events is computed in a single step, further compressing the state space (assuming that a composite component is stimulated in hopes of observing an output). Second, it is sometimes desired to "decorate" a component with auxiliary components such as adapters or monitors (like in the verification models of the MoDeS^3 case study in Sect. 8) without introducing a delay in the observable effect of an event. Furthermore, it is convenient to think in terms of pipelines, which is best expressed with cascade composite components.

One drawback of using cascade composite components is that the outputs of constituent components may overwrite each other if a particular component is run multiple times (but this is still deterministic), and all outputs of all components are emitted in a single event vector. If the temporal unfolding of the different reactions is relevant, it may be more beneficial to use a synchronous composite component. Note that this difference is enhanced in timed systems, as the atomic and instantaneous execution of a cycle implies that feed-forward signals are sent and received *at the same instance of time*, while feedback signals may be delayed in a timed sense as well.

4.6 Event sequences

In the asynchronous-reactive domain, event vectors are substituted by event sequences.

Definition 8 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 $S_\sigma(A)$.

4.7 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 9 An asynchronous component is a tuple $\circleddash = (S, s^0, I, O, \mathcal{D}, T)$:

– 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 $E = I \cup O$.
– $\mathcal{D} : E \rightarrow \{d_1,\ldots,d_n\}$ 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.

Discussion Contrary to synchronous components, the definition of asynchronous components is closest to event-driven finite state machines or the variant of statecharts defined in UML. Although currently not supported by the Gamma Statechart Language, asynchronous components could be implemented directly by statecharts. In Gamma, the current means of defining an asynchronous statechart component are to define a synchronous component containing a statechart and wrap it in an asynchronous adapter.

Note that allowing a nondeterministic transition relation is necessary because the order of output events may not always be specified, e.g., in the case of orthogonal regions. In the case of synchronous components, the order of events does not matter as they are collected in an event vector. The event sequence, however, will be different depending on the internal order of raising events. This phenomenon poses challenges to both verification and code generation and hinders the reproducibility of test cases and counterexamples. Nondeterministic behavior, however, is inherent in the asynchronous-reactive domain anyway.

4.8 Asynchronous adapter

Recall that an asynchronous adapter wraps a single synchronous component and converts it into the asynchronous domain. To do this, the *trigger predicate* with a set of trigger specifications have to be defined (see control specifications in Sect. 3.5). Additional ports may also be defined. Formally, the opportunity to define multiple additional ports and events on them is only a syntactic sugar, as all of them are mapped to the *control event* introduced in the definition below.

Definition 10 An asynchronous adapter for a synchronous component is defined as a tuple $\textcircled{$\sqsubset$} = (\ominus, e_c, \mathit{trig})$:

- $\ominus = (S_s, s_s^0, I_s, O_s, \mathcal{D}_s, T_s)$ is the wrapped synchronous component.
- e_c is the *control event*.
- $C = \{c_1^{t_1}, \ldots, c_n^{t_n}\}$ is the set of clocks, where $c_i^{t_i}$ produces e_c periodically after every t_i.
- $\mathit{trig}\colon I_s \cup \{e_c\} \to \{\top, \bot\}$ is the *trigger predicate* that given an input event, returns whether the wrapped synchronous component must be executed or not.

Semantics The semantics of asynchronous adapters is defined in terms of an asynchronous component. Observed from the environment of the component, an adapter processes input events one-by-one (just like asynchronous components in general), but may not always produce an output. The role of the adapter is to "collect" messages for the wrapped synchronous component, and when a message triggers execution, that is, $\mathit{trig}(e)$ is $\top$, feed the collected messages and emit messages created from the resulting output event vector.

Definition 11 An asynchronous adapter $\textcircled{$\sqsubset$}$ for a synchronous component is itself an asynchronous component $\textcircled{$\sqsubset$}\rangle\textcircled{=} = (S, s^0, I, O, \mathcal{D}, T)$:

- $S = S_s \times v_I$ 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.
- $s^0 = (s_s^0, \bot_I)$, where $\bot_I$ is the empty input vector.
- $I = I_s \cup \{e_c\}$ is the set of input events including the input events of the wrapped synchronous component and the control event. From an input vector v_I, we can derive $v_{I\,s}$ as $v_{I\,s}(e) = v_I(e)$ for every $e \in I_s$.
- $O = O_s$ is the set of output events defined in the wrapped synchronous component.
- $\mathcal{D} = \mathcal{D}_s \cup (e_c \to \{\top\})$ is the domain function of the wrapped synchronous component extended with a mapping that assigns a singleton set to the control event, indicating that it is not parameterized.
- The transition relation is defined as a (nondeterministic) transition function $T\big((s_s, v_I), (e, p)\big) = \{(s_s', v_I')\} \times \Omega$, such that:
  - If $\mathit{trig}(e) = \bot$, then the buffer input event vector is updated such that $v_I'(e) = (e, p)$ and $v_I'(e') = v_I(e')$ for every other $e' \in I$ $(e \neq e')$, and $s_s' = s_s$, while $\Omega = \{\varepsilon\}$ (as the set of possible output sequences) is the empty sequence in this case.
  - If $\mathit{trig}(e) = \top$, then 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 $v_I' = \bot_I$. $\Omega = S_\sigma(\{(e, p) \mid v_O(e) = p, p \neq \bot\})$ (as the set of possible output sequences) is every possible permutation of the "non-null" elements of the output vector.

Discussion The order of messages between two execution-triggering messages is not relevant as long as they do not overwrite each other, so the adapter may store an event vector as a buffer instead of a message queue. In practice, the memory allocated for the input vector of the wrapped component can be reused.

The definition of asynchronous adapters is very flexible. Components like an event-driven finite state machine may be implemented by a synchronous component by declaring no additional control events, but returning $\top$ from the trigger predicate for any event (as seen in Sect. 3.5). With the help of the control event, however, it is also possible to promote the "ticks" of the wrapped synchronous component to its syntactic contract, allowing the environment to execute the component, which is the preferred way of handling even a single synchronous system in Gamma. The definition also allows mixed solutions, e.g., a component may be triggered by any external control event or by any event on one of its ports.

Note that according to the definition, the sequence of output events may be any permutation of the "non-null" events in the output vector of the wrapped component. Although consistent with the definition of asynchronous components, this is rather an underspecification than real nondeterminism—most implementations would raise output events in a fixed order, e.g., when wrapping a cascade composite component.

Finally, note that clocks are not directly handled in the semantics, as the synchronous layer has a notion of logical time only. Nevertheless, clocks are considered as a special source of events occurring spontaneously after every t_i, sending a tick message to the containing component received through the control event e_c (see Sect. 4.11).

4.9 Asynchronous composite component

The syntactic definition of an asynchronous composite component differs from synchronous composite components only in the definition of channels. Since asynchronous components operate with event sequences, it is not a problem anymore if an input event has multiple sources, so there is no restriction on channels other than parameter compatibility.

Definition 12 An asynchronous composite component is a tuple $\textcircled{a} = (\mathbf{C}, I, O, \rightleftharpoons)$:

- $\mathbf{C} = \{\textcircled{=}_1, \ldots, \textcircled{=}_K\}$ is the set of asynchronous components constituting the composite component, each component being $\textcircled{=}_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
- $I \subseteq \hat{I}$ is the set of exported input events, where $\hat{I} = \bigcup_{k=1}^{K} I_k$.
- $O \subseteq \hat{O}$ is the set of exported output events, where $\hat{O} = \bigcup_{k=1}^{K} O_k$.
- $\rightleftharpoons \subseteq \hat{O} \times \hat{I}$ is the set of *channels* that connects inputs and outputs with no restriction apart from parameter compatibility. The set of inputs connected to an output e is denoted by $\rightleftharpoons(e) = \{e' \mid (e, e') \in \rightleftharpoons\}$. We demand that for each $e \in \hat{I}$ and $e' \in \rightleftharpoons(e)$, $\mathcal{D}(e) = \mathcal{D}(e')$. Note that $\rightleftharpoons(e)$ used as a function maps from outputs to inputs, contrary to the notation used in synchronous components, where it mapped from inputs to outputs.

Discussion In asynchronous composite components, events are transferred in messages and processed one-by-one. We assume that components have a message queue where sent but unprocessed messages are stored.

4.10 Environment of the component

The environment of an asynchronous composite component is modeled as follows.

Definition 13 Given an asynchronous composite component $\textcircled{a} = (\mathbf{C}, I, O, \rightleftharpoons)$, its *environment* is a tuple $\textcircled{e} = (E_I^{ext}, E_O^{ext})$:

- $E_I^{ext} = O$ is the input events of the environment that consume the output events of the asynchronous composite component.
- $E_O^{ext} = I$ is the output events of the environment that serve as the input events of the asynchronous composite component.

Discussion The behavior of the environment is considered nondeterministic. In future work, we plan to restrict its behavior with scenario-based contracts.

4.11 Messages and execution traces

The semantics of asynchronous composition can be defined in terms of messages and occurrences. A message is defined in terms of its source and target events and its parameter.

Definition 14 Given an asynchronous composite component $\textcircled{a}$ with its environment $\textcircled{e}$, an asynchronous *message* is a tuple $m = (e_O, p, E_I)$:

- $e_O \in \hat{O} \cup E_O^{ext} \cup C$ is the source output event of the message, possibly coming from the environment or a clock of an asynchronous adapter in the system.
- $p \in \mathcal{D}(e_O)$ is the content of the message.
- $E_I \subseteq \hat{I} \cup E_I^{ext}$ is the set of target input events of the message, possibly targeting the environment.
- If $e_O \in E_O^{ext}$, then $E_I \subseteq I$ and if $E_I \subseteq E_I^{ext}$, then $e_O \in O$, i.e., external messages may arrive through exported input events, while external targets may be messaged from exported output events. If $e_O \in C$ for some asynchronous adapter $\oplus$, then $E_I = \{e_c\}$, where e_c is the control event of $\oplus$. Otherwise, $\rightleftharpoons(e_O) = E_I$, that is, if the message is sent to another component in the same asynchronous composite component, the corresponding inputs and outputs are connected with a channel.

Definition 15 Given a message $m = (e_O, p, E_I)$, let $send(m)$ denote the occurrence of creating the message in response to its source output event and $recv(m, e_I,)$ the occurrence of consuming the message on input event $e_I \in E_I$, thus, raising event e_I. The source component of a message is denoted by $src(m) = \textcircled{=}_k \in \mathbf{C}$ when $e_O \in O_k$ or $src(m) = \textcircled{e}$ if $e_O \in E_O^{ext}$.

Furthermore, let $t = (s, e_I, s', \omega) \in T_k$ be a transition of a constituent component $\textcircled{=}_k$. An occurrence of transition t is a tuple $[t] = (m_I, t, M_O)$, where:

- $m_I = (e_O, p, E_I)$ is the message *triggering* the transition, where $e_I \in E_I$.
- t is the triggered transition.
- M_O is the sequence of *raised* messages such that $|M_O| = |\omega|$ and for every $1 \leq i \leq |M_O|$, $M_O[i] = (e'_O, p', E'_I)$ such that $\omega[i] = (e'_O, p')$ and E'_I obeys Definition 14.

Discussion A message is a runtime object, i.e., it has "object identity." For example, in the *ModesTrack* model (presented in Sect. 3.5 in Fig. 6) event "*Z1.T1ProtocolOutDivergent.go*" creates a message m with itself as the source and

"*Z5.S11ProtocolInCW.go*" as the target, with no parameter (the domain is a singleton set). Raising it again creates another *different* message m' but with the same content. Occurrences, such as message sending, receiving and firing transitions, constitute the observable behavior of an asynchronous system, e.g., sending message m is an observable happening at a specific point in time. Occurrences enable us to define an execution trace, describing the behavior of asynchronous systems.

Definition 16 Given a totally ordered sequence of transition occurrences and message sending and receiving (that is, an *execution trace*), let $\#[t]$, $\#send(m)$ and $\#recv(m, e_I)$ denote the position of the corresponding occurrence in the ordering. The execution trace of an asynchronous composite component must obey the following rules (defining a partial order):

1. *(causality)* $\#send(m) < \#recv(m, e_I)$ for every message $m = (e_O, p, E_I)$ appearing in the execution trace and for every $e_I \in E_I$.
2. *(causality)* $\#recv(m_I, e_I) < \#send(m_O)$ for every transition occurrence $[t] = (m_I, t, M_O)$ appearing in the trace where $m_O \in M_O$.
3. *(message order)* If $\#send(m) < \#send(m')$ such that $src(m) = src(m')$, then $recv(m, e_I) < \#recv(m', e'_I)$ for every e_I and e'_I belonging to the same component $\circleddash_k$ ($e_I \in I_k$ and $e'_I \in I_k$) and assigned to message queues with the same priority.
4. *(message order)* For every transition occurrence $[t] = (m_I, t, M_O)$ and for each $1 \leq i, j \leq |M_O|$ if $i < j$, then $\#send(M_O[i]) < \#send(M_O[j])$.

Furthermore, let $\tau([t])$, $\tau(send(m))$ and $\tau(recv(m, e_I))$ denote the timestamp of the corresponding occurrence according to a common global clock. We require that:

5. *(direction of time)* If $\#occ_i < \#occ_j$, then $\tau(occ_i) \leq \tau(occ_j)$ and if $\tau(occ_i) < \tau(occ_j)$, then $\#occ_i < \#occ_j$.
6. *(periodic ticks)* For each clock $c_i^{t_i}$, there is an infinite number of messages $m = (c_i^{t_i}, \bot, \{e_c\})$ such that $\tau(send(m)) = n \cdot t_i$ for all $n > 0$.

Discussion The first two rules enforce causality: an occurrence cannot happen before another occurrence that caused it to happen. The third rule is a constraint on the implementation of asynchronous systems of the GCL: the communication is demanded to be reliable not only in terms of losing messages (implicitly forbidden by Rule 1), but also in terms of the order of messages. The fourth rule, on the other hand, describes the natural mapping between output event sequences and the generated message sequences. These rules are satisfied when assuming reliable and order preserving message passing in the modeled communication; nevertheless, unreliable communication can be explicitly modeled using additional components (unreliable channel) or as part of the behavior of the communication-related components.

Regarding time, the fifth rule specifies that timestamps are assigned in a monotonic way (but not strictly, i.e., we can force an order on two occurrences with the same timestamp). Finally, the sixth rule defines that clocks emit ticks every t_i time units starting from a designated 0 point of time. Note that (1) whenever the timestamp of a transition occurrence $[t_i]$ differs from the previous transition occurrence $[t_j]$ of the same component, any internal clock variables are assumed to be increased with the same $\Delta t = \tau([t_i]) - \tau([t_j])$ as described in Sect. 4.3 and (2) tick emit messages in a fixed time, but consuming the message can occur anytime later in accordance with the lack of guarantees for execution time and frequency of asynchronous components.

5 Functionalities of the Gamma framework

This section summarizes how the various features and functionalities of the Gamma framework were implemented based on the languages, especially the Gamma Composition Language and its semantics.

5.1 Integrating component models

The integration of external component models (i.e., statechart models from external modeling tools) is realized by mapping these models to GSL models. A separate mapping needs to be defined for each supported statechart variant, taking into account the peculiarities of their syntax and semantics (e.g., the rules for mapping Yakindu statecharts can be found in [24]). Note that the GSL is prepared for supporting different statechart semantics by offering annotations. GSL, as a common representation of component statecharts, allows the use of the functionalities of the Gamma framework by integrating components designed in various external tools (currently, Yakindu and MagicDraw are supported, integrating other UML/SysML tools would be only a technological challenge).

While the GSL would be suitable for most statechart-like formalisms, it has certain limitations as well. For example, internal signals commonly used in UML/SysML models are not supported, i.e., there is no direct way to synchronize parallel regions of a composite state apart from *join* transitions. Also, the support for timed behavior is limited to deterministic timeouts that are not sufficient to model nondeterministic completion times for example. Some of these limitations are highlighted in the experimental case study in Sect. 6.

Nevertheless, the tool architecture allows the extension of the framework with arbitrary new formalisms exposing the behavior described in Sect. 4.3. We are also constantly working on extending the GSL (e.g., with a richer action language), but the details of atomic component implementations are out of the scope of this paper.

5.2 Model validation

Validation takes place at the level of atomic component models and during the composition of components. It is realized by specifying ill-formedness constraints in the form of model patterns that are evaluated at design time. Validation regarding composition realizes the constraints presented in Sect. 3, that is, the checking of model imports, port bindings, values bound to parameters of components, channel constructions, execution of components in cascade models, and attribute values as well as event references in control specifications and message queues in asynchronous adapter models.

5.3 Code generation

The Gamma framework supports automatic source code generation from Gamma models. Currently, Java code is generated.

In the case of atomic components, Gamma reuses the code generators provided by the integrated external modeling tools, e.g., Yakindu.

Interface definitions are generated from Gamma interfaces and are realized by the corresponding port objects of the component implementations. As a result, users can interact with the component objects through the well-defined, familiar interfaces.

Composite component implementations are generated from Gamma composite components defined in GCL. They wrap the atomic component implementations and subordinated composite components and construct the required connections (channels) between them. Thus, wrapped components are able to communicate with each other by dispatching and receiving event objects. Note that the generated composite component implementations are independent from the atomic component implementations and communicate with them only via a well-defined interface. Thus, atomic component implementation other than the currently supported Yakindu can be included without the rework of the Gamma code generator, and even different component implementations can be executed side by side in the same composite system.

It is essential that the behaviors of composite components conform to the rigorous semantics of the GCL introduced in Sect. 4. This is realized using different programming constructs in each component type. Synchronous composite and cascade components wrap atomic components and an external tick (an explicit method call) is implemented to trigger their execution, which is also transferred to their contained components, eventually reaching the atomic components. Asynchronous adapters, however, rely on concurrent components that run continuously without external ticks. This way, asynchronous adapters provide the ticks necessary for the execution of their wrapped synchronous components. Asynchronous composite systems have two different implementations. One of them uses threads hosted in a single process to implement the composed asynchronous adapters (each running in its own thread). The other one is based on inter-process communication. Communication between asynchronous Gamma processes to be deployed to separate nodes in a local network is realized using the DDS standard.[16]

5.4 Formal verification

Formal verification in the Gamma framework is performed by the external model checker UPPAAL (also, the architecture supports the integration of additional model checkers). The integration is achieved by automatically mapping the Gamma composite system model to a network of timed automata, the input formalism of UPPAAL. The queries representing the properties to be checked on the model are either given directly as UPPAAL temporal logic formula or constructed using fillable patterns (for the most frequent safety and liveness properties).

For each atomic statechart component, a set of automaton templates is generated (one template for each region) that synchronize with each other using channels [24]. These templates are instantiated in accordance with the component instance definitions. Similarly to source code generation, it is important to ensure that the generated formal model behaves according to the semantics of the composite system. This is supported by creating auxiliary model elements responsible for conducting interactions among these independently transformed atomic components, as follows.

For each synchronous composite and cascade component, a scheduler automaton is generated that controls the execution of instantiated atomic components. Each input event of a synchronous composite component is handled through a vector consisting of two boolean variables, *toRaise* and *isRaised*. An input event can be raised by setting the *toRaise* variable to true (by the environment automaton modeling external events or by other component automata). At the start of each cycle, its value is copied into the *isRaised* variable, which is read by the automaton during execution. Contrarily, each input event of a cascade component is modeled as a single variable that can be written and read arbitrarily, making in-cycle communication possible.

For each asynchronous adapter, an executor automaton together with message and message queue data structures

[16] The DDS-based asynchronous Gamma code generator was realized by freshman university students, which demonstrates the easy extensibility of Gamma code generators.

is generated. Message insertion/removal to/from queues are implemented by functions, which are called by the executor automaton. The removed messages are transferred to the wrapped synchronous component. Also, if a message triggers the execution of the wrapped component, its scheduler automaton is initiated. To model clocks, an automaton is created that generates clock messages in defined intervals.

In the case of asynchronous composite components, an asynchronous scheduler automaton is introduced that initiates the execution of the executor automata of the composed adapter components.

5.5 Test case generation

Gamma supports model-based testing by using the integrated model checker to generate tests. The idea is to construct queries for the model checker and interpret the resulting traces (witnesses or counterexamples generated by the model checker on the basis of the queries) as test cases [25,26]. If the queries are constructed to check the reachability of states (fireability of transitions), then the resulting test suite is able to provide state coverage (transition coverage, respectively). The resulting traces are mapped to the GTL, and test cases (currently JUnit) are generated.

Note that the execution of the generated JUnit test cases on the (generated) code of a composite system validates the following functionalities of the Gamma framework: (1) mapping component models from external tools to Gamma, (2) transformation of Gamma models to a back-end model checker, (3) code generation from composite components, (4) implementations of integrated atomic components, (5) construction of executable test cases. This kind of validation of the Gamma workflow is essential, as the framework builds on model transformations that are hard to verify. Nevertheless, each witness or counterexample generated by formal verification can be validated, demonstrating that the framework is well-functioning or a real fault is detected. Note that if certain behavior is absent, e.g., a state is not reachable, this technique cannot be utilized.

6 Case study—Simple Space Mission

This section introduces an example model from the aerospace domain which we use as a case study to demonstrate the applicability of Gamma to describe SysML behavioral models. The example model was proposed by NASA in the context of the OpenMBEE[17] framework. The goal of OpenMBEE is to create a common model repository to facilitate tool integration, so the ability to handle models in the scope of this project can raise the relevance of any model analysis tool.

[17] https://www.openmbee.org/.

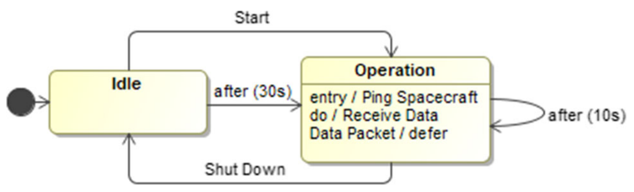


Fig. 7 The statechart model describing the behavior of the ground station component

In the case study, we investigate how SysML models with composite structure state machines and activity diagrams can be mapped to the composition language of the Gamma framework. We use the languages presented in this paper to manually model a simple space mission that describes communication between a satellite and a ground station.[18]

6.1 Modeling of the system

The SysML model describes how a satellite communicates with a ground station as well as the state of the battery of the satellite. The state-based behavior of the system can be seen in Figs. 7 and 8, while the more complex activities are depicted in Figs. 9 and 10. The equivalent, purely state-based Gamma models[19] that we created manually based on the aforementioned models are visualized in Figs. 11 and 12.[20]

The main challenge of mapping this model came from the mixing of state and activity-based behavior. We found that actions with time constraints can be modeled as states with timeouts (although GSL currently does not support nondeterministic timeouts, so we chose a fixed value), so each thread in the activities could be modeled with orthogonal regions. The processed model already followed the convention of handling external signals at the state-based level while parallelly executed activities communicate with internal signals. This internal communication could be mapped to guarded transitions affecting the parent state of the communicating orthogonal regions. Observe the handling of low battery in Figs. 9 and 10 with internal signal *Battery Low* in Fig. 11.

Since the original model was simulated with Cameo Simulation Toolkit[21] with discrete time steps, we modeled the composition with a synchronous semantics. The model of the composite component can be seen in Fig. 13.

[18] The initial SysML models can be found at: https://github.com/Open-MBEE/OMGSpecifications.

[19] The Gamma models can be found at: https://github.com/FTSRG/gamma/tree/master/examples/hu.bme.mit.jpl.spacemission.casestudy.

[20] The visualization of the Gamma models was generated using PlantUML.

[21] https://www.nomagic.com/product-addons/magicdraw-addons/cameo-simulation-toolkit.

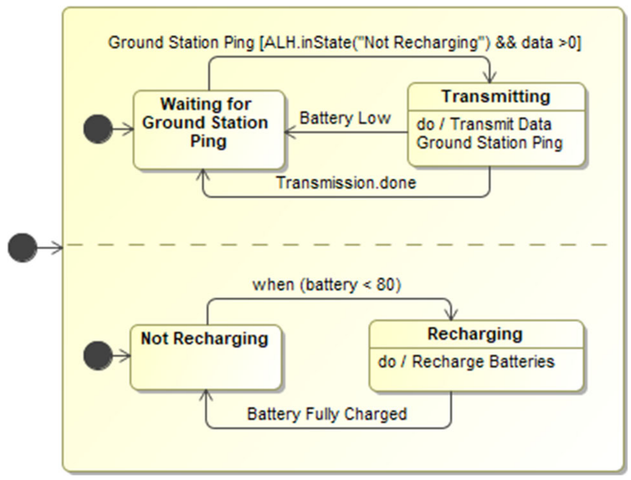

Ground Station Ping [ALH.inState("Not Recharging") && data >0]
Waiting for Ground Station Ping
Battery Low
Transmitting
do / Transmit Data Ground Station Ping
Transmission.done
when (battery < 80)
Not Recharging
Recharging
do / Recharge Batteries
Battery Fully Charged

Fig. 8 The statechart model describing the behavior of the spacecraft component

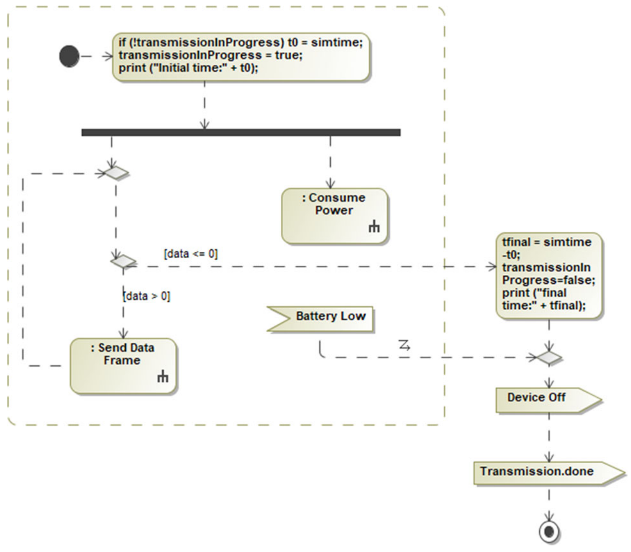

if (!transmissionInProgress) t0 = simtime;
transmissionInProgress = true;
print ("Initial time:" + t0);
: Consume Power
[data <= 0]
[data > 0]
tfinal = simtime -t0;
transmissionInProgress=false;
print ("final time:" + tfinal);
Battery Low
: Send Data Frame
Device Off
Transmission.done

Fig. 9 The activity diagram describing the data transmission process of the spacecraft component

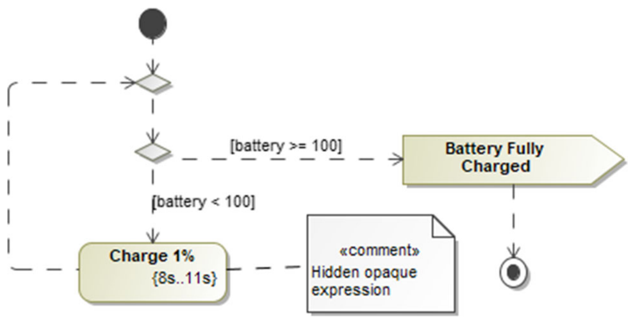


Fig. 10 The activity diagram describing the battery recharge process of the spacecraft component

6.2 Results and conclusion

To check the conformance of the Gamma model with the original SysML model, we have generated a set of state-covering tests using the test generation capabilities of the framework based on the integrated UPPAAL model checker. All the traces were consistent with the behavior described by the SysML model.

This case study showed that SysML models built with certain conventions can be mapped to Gamma models, where they can benefit from the precise composition semantics as well as the code/test generation and verification capabilities. Our approach was manual, because the mapping is not trivial, and some constructs do not fit with our semantics. A notable example is using actions with time constraints or event receptions in activities describing effects of transitions or entry/exit actions: our semantics explicitly forbids waiting during the execution of transitions. We believe this modeling practice should be followed in other tools as well, but the definition of the corresponding conventions is out of scope of this paper.

Nevertheless, this case study provided valuable insights into the mapping of SysML models to Gamma, which was automated for a subset of modeling elements and successfully used in the case study described in the next section.

7 Case study—Orion

This section demonstrates a formal analysis approach for communication protocols using the Gamma framework based on our previous work [27]. This approach supports (1) the construction of protocol participant models as well as channel models with different failure modes, (2) the composition of protocol participant and channel models to form composite system models and (3) model checking on the system models with automatic back-annotation of the results. The process is presented in the context of Orion, a master–slave communication protocol under design targeted to be used in the railway industry. [22]

In this case study, we demonstrate the practical usability of the Gamma framework in the context of an industrial problem. The case study highlights the differences between composition modes to support different potential execution platforms and provides measurement results regarding the verification of the robustness properties of the modeled protocol.

[22] All Gamma models presented in this case study can be found under the following link: https://github.com/FTSRG/gamma/tree/master/examples/hu.bme.mit.gamma.industrial.protocol.casestudy.

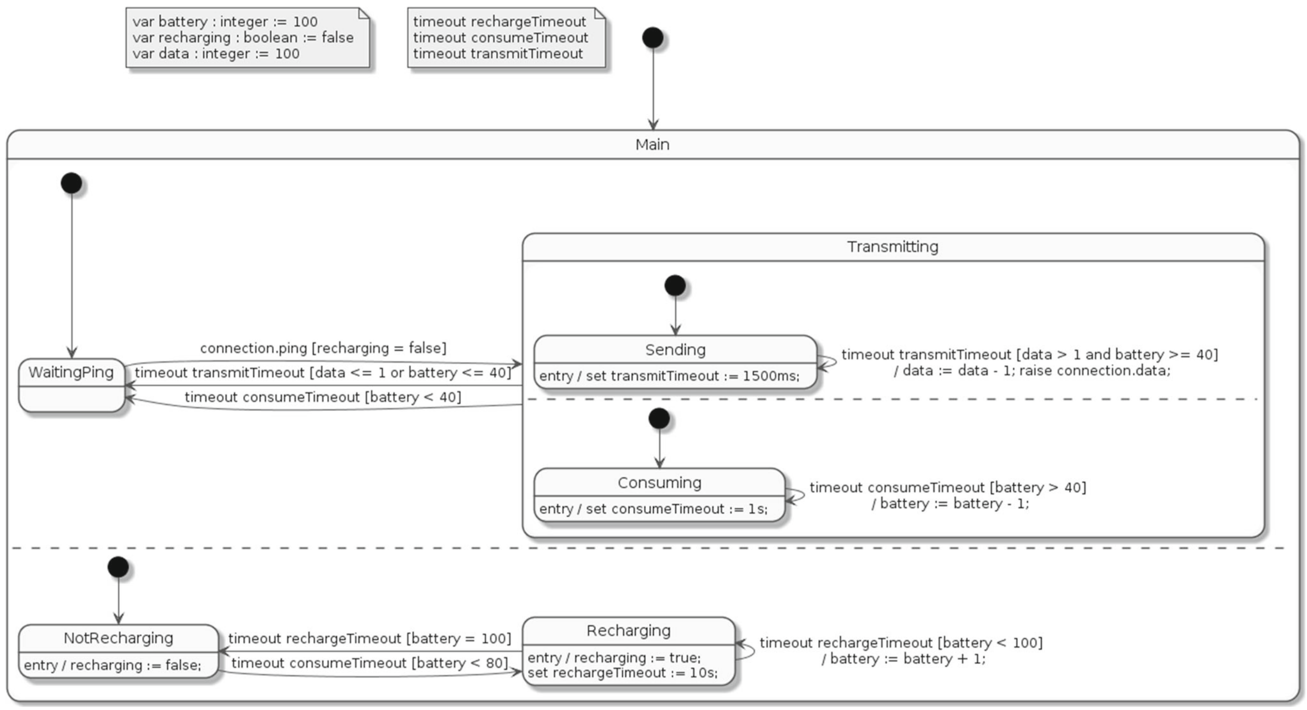

var battery : integer := 100
var recharging : boolean := false
var data : integer := 100
timeout rechargeTimeout
timeout consumeTimeout
timeout transmitTimeout
Main
Transmitting
WaitingPing
connection.ping [recharging = false]
timeout transmitTimeout [data <= 1 or battery <= 40]
timeout consumeTimeout [battery < 40]
Sending
entry / set transmitTimeout := 1500ms;
timeout transmitTimeout [data > 1 and battery >= 40] / data := data - 1; raise connection.data;
Consuming
entry / set consumeTimeout := 1s;
timeout consumeTimeout [battery > 40] / battery := battery - 1;
NotRecharging
entry / recharging := false;
timeout rechargeTimeout [battery = 100]
timeout consumeTimeout [battery < 80]
Recharging
entry / recharging := true; set rechargeTimeout := 10s;
timeout rechargeTimeout [battery < 100] / battery := battery + 1;

Fig. 11 The constructed Gamma statechart model with inlined activity diagrams describing the behavior of the spacecraft component

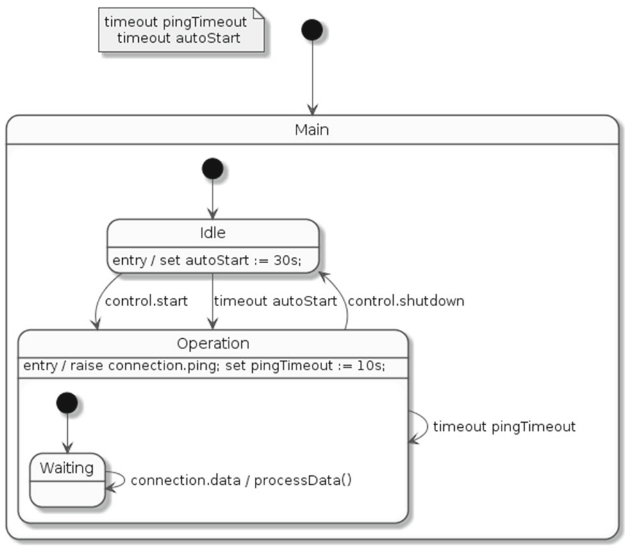


Fig. 12 The constructed Gamma statechart model with inlined activity diagrams describing the behavior of the ground station component

7.1 Modeling of the communication protocol

This section introduces the modeling process of the proposed analysis approach in the context of Orion.

```
sync Mission [
  port control : requires StationControl
] {
  // Declaring mission components
  component station : GroundStation
  component satellite : Spacecraft
  // Binding the control port to the port of station
  bind control -> station.control
  // Connecting the satellite and the station
  channel [satellite.connection] -o)-
    [station.connection]
}
```

Fig. 13 The GCL model describing the mission with a ground station and a satellite

7.1.1 Protocol participants

Orion is a master–slave communication protocol where the establishment of a connection between two participants is always initiated by a master and the connection request is either accepted or rejected by a slave. Both the master and the slave participants were designed on the basis of statecharts in MagicDraw and have the same events (commands and messages).

The initial state of the *master* statechart (depicted in Fig. 14) is *Closed*. When initiating a connection with the slave, it goes to state *Connecting* and waits for a response. Upon a positive response, it goes to state *Connected* whereas upon a negative one or after a certain timeout (*TConn* sec), it goes to state *Closed*. Communication can take place in state *Connected*.

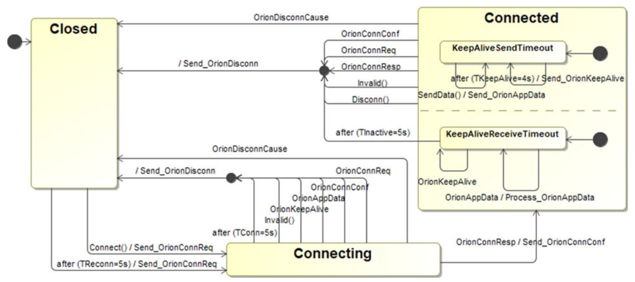


Fig. 14 The statechart model describing the behavior of the master component

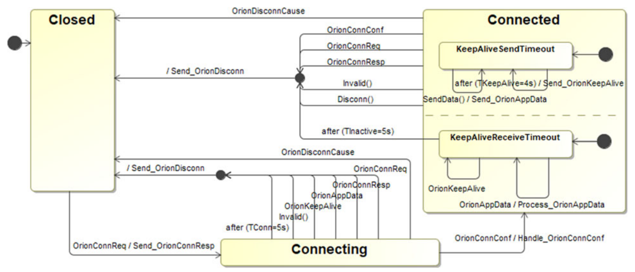


Fig. 15 The statechart model describing the behavior of the slave component

The *slave* statechart (depicted in Fig. 15) is similar to the master.

The models can be automatically transformed to the GSL using the model transformers of Gamma, in which they can be validated based on statechart-related well-formedness rules [24]. According to the validators of Gamma, the presented statechart models are well-formed.

7.1.2 Channel models

During the modeling of communication between protocol participants, several failure modes of event transmission can be considered [28]. In this case study, we defined five channel models in Yakindu: one *ideal channel*, three models describing loss of events failure modes (*bursty message losing channel*, *arbitrary message losing channel* and *timed message losing channel*) and one model related to delay of events failure mode (*delay channel*). They are introduced in [27] in detail, here we present only the simplified version of the *bursty message losing channel* model.

Figure 16 depicts the *bursty message losing channel* model, which models a channel that can lose a given amount (*LOST_MESSAGE_MAX*) of subsequent incoming events. It has two states, *Operating* (initial state) and *MessageLosing*. If the model receives a certain event in state *Operating*, it either forwards the event to its output, or (if there has been no failure before) goes to state *MessageLosing* without forwarding the event. In state *MessageLosing*, the specified amount of events are absorbed before going back to state *Operating*.

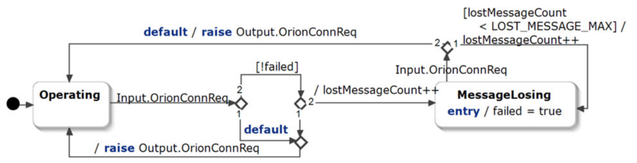

[lostMessageCount < LOST_MESSAGE_MAX] / lostMessageCount++
default / raise Output.OrionConnReq
[!failed]
Input.OrionConnReq
Operating
Input.OrionConnReq
/ lostMessageCount++
MessageLosing
entry / failed = true
default
/ raise Output.OrionConnReq
1
2

Fig. 16 Excerpt from the statechart model of the *bursty message losing channel*

```
[sync | cascade | async] OrionSystem [] {
  // Declaration of components
  component master : OrionMaster
  component m2S : Channel
  component slave : OrionSlave
  component s2M : Channel
  // Connecting component ports via channels
  channel [master.SendOrion] —o)— [m2S.Input]
  channel [m2S.Output] —o)— [slave.ReceiveOrion]
  channel [slave.SendOrion] —o)— [s2M.Input]
  channel [s2M.Output] —o)— [master.ReceiveOrion]
}
```

Fig. 17 The GCL model of protocol participants and channel models

Note the nondeterministic nature of this model: the loss of subsequent events can start on any incoming event. Also, this model includes behavior only for a single event (*OrionConnReq*), nevertheless, additional events in the complete model are handled analogously.

7.1.3 System models

We analyzed the behavior of the Orion protocol considering different channel failure modes and different execution modes of the participants. Therefore, for each channel model we defined *synchronous*, *cascade* and *asynchronous* composite Gamma models, which differ only in the execution mode, the components and their connections are the same. We focused on the time-driven behavior and the events of the Orion protocol in the master and slave components.

Figure 17 describes the GCL model the variations of which were used with different channel models and execution modes. It consists of a master component, a slave component, and two channel components that connect the output and input ports of the protocol participants. The concrete models differ only in the first keyword of the model that can be either *sync*, *cascade* or *async*. All in all, fifteen composite models were defined, five (as there are five channel models) for each composition mode. In the asynchronous composite models, message queues with capacity 2 were used.

7.2 Analysis of the communication protocol

We analyzed liveness properties of the system models introduced in Sect. 7.1.3, that is, the reachability of system states with different channel models using the integrated UPPAAL model checker. The analyzed properties (formalized in CTL [29]) are the following.

Table 2 Average time of verifying P_2 in different system models

	Synchronous (s)	Cascade (s)	Asynchronous (s)
Ideal channel	0.01	0.01	0.4
Bursty message losing channel	0.4	0.3	1.0
Arbitrary message losing channel	2.0	1.7	140.4
Timed message losing channel	0.02	0.02	0.1
Delay channel	4.3	4.1	7.3

Table 3 Average resident/virtual memory peaks during the verification of P_2 in different system models

	Synchronous (Mb)	Cascade (Mb)	Asynchronous (Mb)
Ideal channel	10/48	9/47	12/53
Bursty message losing channel	13/55	12/53	23/72
Arbitrary message losing channel	16/60	14/56	143/314
Timed message losing channel	11/50	11/50	12/53
Delay channel	76/172	61/143	100/220

P_1 The system *can reach* a state in which both the master and the slave are in state *Connected*: `EF master.Connected && slave.Connected`.

P_2 The system *must eventually reach* a state in which both the master and the slave are in state *Connected*: `AF master.Connected && slave.Connected`.

P_1 describes the reachability of the desired operational state from the initial state in the system and means that the master and slave models do not contain fundamental design faults that hinder the correct operation of the protocol. P_2, as a strong robustness property means that the protocol is always able to recover despite the specified failure mode of the channel.

According to the verification executed in Gamma, P_1 *holds* in the case of every system model introduced in Sect. 7.1.3. However, the analysis results regarding P_2 revealed important constraints on the execution frequency of components in each composition mode: since the protocols have (real-time) timeouts, the fulfillment of the property depends on the execution frequency of the system components in the case of each channel model and each composition mode. Using the model checking and automatic back-annotation functionalities of the framework, we analyzed the necessary scheduling order and frequency of components with several parameters of the channel models. The property was fulfillable in every system model with adequate execution characteristics. The detailed constraints on the execution characteristics to fulfill the property can be found in [27].

To extend [27] and provide additional insight into the verification capability of Gamma, we measured the time (see Table 2) and the memory consumption (see Table 3) of verifying P_2 with respect to the defined system models. In the cases of the *bursty* and *arbitrary message losing channel* models the value of the *LOST_MESSAGE_MAX* parameter was 5. In the case of the *timed message losing channel*, the values of the *S* and *E* parameters were 4 and 9, and for *delay channel* the value of the *T* parameter was 1. The execution frequencies were set in accordance with the constraints mentioned above. We ran and averaged 10 measurements for each system model.

7.3 Results

The measurement results show that both verification time and memory consumption in the case of cascade models is slightly less than in the case of synchronous models. This is the result of the model transformation implementation as in synchronous models each event is mapped to two variables, whereas in cascade models a single variable is defined. Also, in the case of synchronous models event transmission between the master and slave components requires multiple cycles contrary to cascade components, which can also result in higher memory consumption and verification time.

As expected, asynchronous models are significantly harder to verify than synchronous and cascade models due to the additional message queue structures and scheduler components. The difference is markable in the case of the *arbitrary message losing channel* where there is a 70-fold difference in verification time compared to the synchronous model.

7.4 Conclusion

This case study demonstrated that the composition semantics proposed by Gamma can indeed be used in practice: even though the semantics introduced in Sect. 4 assumes instantaneous, reliable and order-preserving message passing, the language is applicable to practical problems by introducing (fault) models describing physical phenomena, e.g., loss or delay of events.

The three composition semantics cover diverse execution and communication modes of the composed components, which could be used to model different execution platforms in the case study. As Gamma supports the automatic import of models defined in integrated modeling front-ends, we did not have to manually transform the already existing component models, which greatly reduced the effort required for the approach. Also, the measurement results show that the formal verification capabilities of the framework are applicable in practice.

Furthermore, the case study revealed the need for potential *platform models* in the case of timed component models where the execution frequencies of the component(s) under verification can be specified. Consequently, we have extended the verification capability of the framework with this feature.

8 Case study—MoDeS^3

This section presents a case study in the context of the MoDeS^3 project (introduced in Sects. 2.1, 3), where we used the Gamma framework in the full design and verification process of the safety logic preventing collisions on the tracks. We relied on the flexible languages of the framework in both the design and verification tasks.

Here, we present the models and (manual) methods we used to incrementally transform a high-level specification into the final implementation of the safety logic controlling train movement on sections (turnouts have been added in a later phase which we do not discuss here). The primary requirement of this logic is the following: *“Two trains must not be positioned on the same section at the same time.”* This demonstration focuses on how composition supported by Gamma can help in the transformation of an initial specification, the execution of verification tasks, as well as the effect of different interaction semantics that could not be modeled directly in SysML-based tools alone.[23]

8.1 Designing the controller with a simplified MoDeS^3 track setup

The core functionality of the safety logic is to prevent that a train approaches another train in such a way that both of them occupy the same section. To design and check this functionality, we used a circular track setup with 8 sections (altogether *S01* to *S08*). Also, we divided the track into two zones (supervised by two different physical controllers), each zone consisting of 4 sections. We used two trains that are initially situated on opposite sections of the circular track (*S01* and *S05*). This arrangement covers all the interesting scenarios regarding train movement and communication between the zone controllers.

[23] All models presented here can be found at: https://github.com/FTSRG/gamma/tree/master/examples/hu.bme.mit.gamma.modes3.casestudy.

The safety logic implements the following safety concept: sections before and after an occupied section are considered as an “aura” of the train; when collision of auras is detected then the related section has to be disabled (stopping the train on that section). We modeled and analyzed two operation modes differing in the way of communication between the track and the train.

- *Restrictive operation mode*: initially, all sections are *disabled*; a train is *not* allowed to move on a particular section until the section gets *enabled* and the controller sends an *enable* event to the train. This is analogous to supplying power to the tracks in a section on demand.
- *Permissive operation mode*: initially, all sections are *enabled*; a train is allowed to move on a section until the section gets *disabled* and the controller sends a *disable* event to the train. This is analogous to cutting the power supply in case of danger.

The restrictive operation mode can be adequately modeled with synchronous composition (discussed in Sects. 8.2–8.4) using the features of synchronous communication: in such a model, the controller can *always* observe the location of the trains before their movement and enable the connecting sections if it is safe to do so. Synchronous composition is convenient to ensure a lock-step execution of the controller and the trains, which is necessary for this strategy. An inconvenient scenario for this mode is the unnecessary waiting of trains if the controller is too slow.

We chose to model the permissive operation mode with asynchronous composition (see Sect. 8.5) to demonstrate the different characteristics of asynchronous communication compared to synchronous communication. Without further restrictions, this composition mode will let components run independently with any frequency. Even though this strategy will not stop trains unnecessarily, a slow controller may fail to react in time when there is a danger of collision. This will be revealed by verification, too, showing again that in the case of asynchronous systems, practical problems will require further constraints on timing.

All safety logic components, both for restrictive and permissive operation modes, have the same number and type of ports (i.e., the same interface from the point of view of the environment): *Train* ports, on which the *occupy* and *unoccupy* events for particular sections are received (modeling sensors), and *SectionControl* ports, on which *enable* and *disable* events are transmitted (modeling e.g., power to the train). Recall that these interfaces are presented in Fig. 5a.

To design the controller for this layout, we start from a high-level specification describing the overall behavior of the safety logic (Sect. 8.2). In this model, all information is assumed to be available locally and we can focus on the main logic. Accordingly, we verify that the concept of "auras" is sufficient to prevent collisions at this level. As we intend to use two physical controllers, we split the model into zones and introduce a communication protocol to obtain non-local information about the track (Sect. 8.3). The correctness of the protocol is verified by proving the *conformance of the inputs and outputs* with the original specification by checking *trace equivalence*, which is achieved by modeling a special composite component only for verification purposes. Finally, we split zones into components responsible for a single section to facilitate the component-based modeling of any kind of zones (Sect. 8.4). The correctness of this step is verified as the previous one, and the two results together yield a proof that the model hierarchically assembled from section controller components into zones and then the whole logic also prevents collisions, as proven on the high-level specification model.

8.2 High-level safety logic specification

The *high-level safety logic specification* is modeled as a single atomic synchronous component, just like another model simulating two trains. The *high-level system model* composes these components into a cascade composite component for verification purposes. As the train model receives commands from the environment about how to simulate trains (move them forward or backward), then notifies the safety logic model about the occupation and unoccupation of sections, it is reasonable to execute the train model and the safety logic model in this order (described by the execution list of the enclosing cascade composite component). Replies of the safety logic are processed in the next cycle, which is again reasonable to assume, as the response might take time to compute.

The high-level specification is defined in terms of a single state and two boolean variables for each section, one of which denotes whether a particular section is *occupied* by a train (sensors), whereas the other one denotes whether it is *enabled* (actuation). The transitions describe what action should be taken when trains move around the track, i.e., what signals shall be sent to actuators in response to sensor input.

The train model simulates two trains on the track representing their motion on the subsequent sections. This model is symmetrical in many ways, as the two trains behave the same way, and a train behaves analogously on the first, second, etc., sections when moving forward or backward.

Each train has a *TrainControl* port as well as (according to the restrictive operation mode of the safety logic) an *enabled* variable that keeps track of whether the train is allowed to move or not. We assume that trains have nonzero length, but are shorter than sections, so they move onto the next section gradually, thus, in an intermediate step it is situated on both sections (but never more than two). The position of each train is encoded in the following way: if variable *position* is set to a number with a single digit, it means the train is situated on a single section. If the position is set to a number with two digits, e.g., 12, it means the trains is situated on two sections, *S01* and *S02*, at the same time.

Verification The *high-level system model* is mapped to UPPAAL, and a formal query representing the safety requirement is verified. In the case of this model, there is no single erroneous state to avoid. Instead, the following statement is formalized: *"It is impossible to reach a state in which more than one train occupy the same section."* In the subset of CTL [29] supported by UPPAAL:

```
A[]!((position1Oftrain == 81 || position1Oftrain == 1
      || position1Oftrain == 12) && (position2Oftrain
      == 81 || position2Oftrain == 1 ||
      position2Oftrain == 12) && ...)
```

Recall the encoding of the states of the trains in the train model (Sect. 8.2) and note that each section is checked one by one for both trains. This safety property holds on the *high-level system model*, that is, two trains can never occupy the same section.

8.3 Medium-level safety logic model

The *high-level safety logic model* cannot be deployed directly to the simplified MoDeS^3 system as it consists of communicating microcontrollers (one for each zone). Therefore, we split the model into two *medium-level zone models* composing the *medium-level safety logic model* as a *synchronous composite component*. At this level, synchronous composition is a suitable choice as there should be no ordering between the zones, and more importantly, signals between the zones *may be delayed* (logically, we do not use timing in these models).

The *medium-level zone model* is a Gamma statechart similar to the high-level specification, but modeling only four sections and introducing two *Protocol* ports to support communication on both ends of the zone (CCW and CW, which stand for clockwise and counter-clockwise, respectively). Contrary to the *high-level safety logic model*, communication is necessary in this model to learn whether neighboring sections on the edges of zones are occupied.

Verification At this level, the verification task is to prove conformance between the high-level and the medium-level safety logic models by checking trace equivalence. To this end, we design a new *oracle component* responsible for comparing the outputs (*enable* or *disable*) of the two variants and moving to an *Error* state if inputs differ. Therefore, the goal is to prove that this state is unreachable.

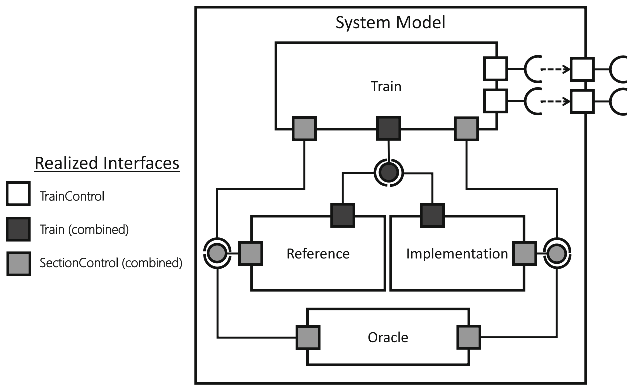


Fig. 18 A schematic figure about the *system model to check the conformance of the inputs and outputs* of components. Rectangles represent component instances, squares represent ports of components, which realize interfaces either in provided mode (lollipop) or required mode (socket). Channels are represented as the connection between these. Note that ports realizing the *Train* and *SectionControl* interfaces are depicted jointly

To check conformance, we define a cascade composite model that composes the *train model*, the *high*- and *medium-level safety logic models* (as reference and implementation, respectively) and the *oracle model* into a single system (depicted schematically in Fig. 18). Signals from the train model are sent to both variants of the safety logic model, while their outputs are replicated to pass to the oracle in addition to the train. Note that cascade composition is again a natural choice here, as there is a clear pipeline-like ordering between the components, and the oracle should process outputs in the same turn.

Once we have the full *system model*, it is transformed to UPPAAL, and the following query is verified:

```
A[]!(P_mainRegionOfStatechartOfOracle.Error)
```

The query is the formalization of the following statement: *"State Error of component oracle is never assumed."* The query holds, this way it is impossible for the two safety logic models in the *system model* to produce inconsistent section control events: the refinement is correct.

8.4 Low-level zone model

The previous models are useful for the high-level description and verification of the safety logic for the specified track setup. However, to support arbitrary zone setups, it is worth defining a *section model* that controls a single section and compose its instances using synchronous composition to define controllers of the zones.

Accordingly, the *medium-level zone model* (Sect. 8.3) is extended by instantiating the particular section model four times and composing its instances in a synchronous composite component, the *low-level zone model*. Note that the cascade composition would not be favorable here as it is not possible to define a justified execution order of the components, and they communicate with signals that should not be prioritized. The protocol for communication between the section models is similar to communications between zones.

Verification Similarly to the method used in Sect. 8.3, the conformance between the *medium*- and *low-level zone model* can be verified by constructing a *system model* with an oracle model and performing model checking on that. As before, the two models turned out to be conformant.

Figure 19 depicts the analyzed conformance conditions between the *high*- and *medium-level safety logic model* as well as between the *medium*- and *low-level zone models*. Note that existence of the conformance relations between these models also implies that conformance holds between the *medium*- and the *low-level safety logic model* composing two *low-level zone components* synchronously. Considering these results, in addition to the results presented in Sects. 8.2 and 8.3, the design process for the restrictive operation mode of the safety controller with respect to the safety requirement was proved to be correct.

8.5 Asynchronous medium-level safety logic model

In this section, we investigate the effect of using the permissive operation mode of the safety controller. We use the *medium-level zone model* wrapped in an asynchronous adapter and instantiated two times for the two zones in the asynchronous composite *asynchronous medium-level safety logic model*. A variant of the *train model* is also defined to support asynchronous operation (with an adapter), and the two asynchronous components compose the *asynchronous system model* on which formal verification is carried out with respect to the safety requirement. The train model variant here is different from the one used in the restrictive operation mode in that it does not expect an explicit enabling signal from the safety logic to move, but an explicit disable signal can stop it.

Verification After transforming the resulting model to UPPAAL and evaluating the query presented in Sect. 8.2, we receive a counterexample, in which the model checker gives an execution trace leading to a state where two trains occupy the same section (that is, there is a collision). The problem is the following: the *asynchronous train model* is scheduled multiple times and moves the simulated trains toward each other, whereas the safety logic is not scheduled at all. This is a valid trace that conforms to the asynchronous semantics of Gamma, according to which there is no guarantee on the execution frequency of asynchronous components. As for real life, situations where the safety logic controllers are not executed fast enough compared to the moving of the trains, although unlikely, can indeed happen. The problem could be mitigated by defining constraints on the execution frequency of the components.

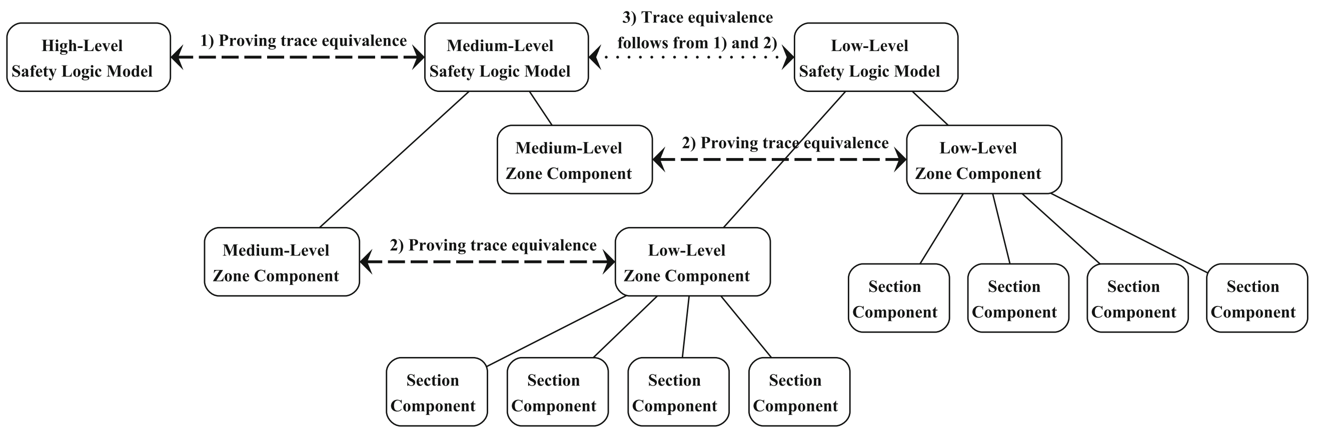

High-Level Safety Logic Model
1) Proving trace equivalence
Medium-Level Safety Logic Model
3) Trace equivalence follows from 1) and 2)
Low-Level Safety Logic Model
Medium-Level Zone Component
2) Proving trace equivalence
Low-Level Zone Component
Medium-Level Zone Component
2) Proving trace equivalence
Low-Level Zone Component
Section Component

Fig. 19 A schematic figure about the results of the verification of trace equivalence conditions

8.6 Conclusion

This case study demonstrated how the capabilities of the framework could be used to support the incremental design and verification of a distributed safety logic. Our approach was manual and based on proving conformance between system models with trace equivalence. The process was facilitated by several features of GCL.

- In GCL, two system models are interchangeable if their "interfaces" match (same number and type of ports); their internal definitions from a composition point of view are irrelevant.
- GCL supports hierarchical composition: the derivation of new system models by introducing additional (composite) components inside the model is a fundamental feature.

As GCL supports both synchronous and asynchronous composition semantics, the analysis of different system operation modes (execution and interaction modes of the contained components) was feasible. The need for specifying execution frequencies regarding components during verification also emerged in this case study (see Sect. 7 and [27,30] for additional examples for the necessity of specifying constraints on execution frequency).

The case study showed that formal verification is applicable even in the case of larger, non-trivial systems if we can apply convenient abstraction and conformance-proving techniques. However, as this case study demonstrated, the manual execution of such a process can be cumbersome. Consequently, we aim to introduce support for such automated techniques in the Gamma framework.

9 Related work

As related work, we cover in detail languages and tools that provide (1) a composition language for component-based design with (2) precise formal semantics and (3) formal verification support for behavioral properties. There are similar approaches to ours, such as [31,32], where a general CPS modeling language was developed to semantically integrate the models coming from various CPS design tools. The introduced modeling language was formalized; however, the tool lacks formal verification support. The RoboChart modeling tool is introduced in [33] that uses statechart models tailored to the robotic application domain. In RoboChart, a formal semantics helps engineers verify the designed systems. RoboChart was omitted from the comparison as it primarily targets a special domain. Stateflow is a commercial state-based modeling tool: authors in [34] developed a tool based on UPPAAL to formally verify behavioral models. The commercially available Simulink tool can generate source code from the verified Stateflow models. We omitted this solution from the comparison due to the fact that the modeling tool is commercial. Nevertheless, Stateflow models could also be represented in the Gamma Statechart Language.

Clafer [35] is a tool for modeling structure, behavior and variability of systems. Clafer has a formal core modeling language to represent the system design and specification patterns. However, it does not provide formal verification support. Another approach [36,37] uses SysML to model the security aspects of systems and ProVerif is used to verify security properties: this approach is omitted from the comparison due to the lack of generality.

Other languages and approaches, such as [38–42] capture the architecture, mainly focusing on the interfaces, connectors and their relations in systems without defining the behavior of the components. In [43], model-driven techniques and tools are integrated successfully with standard-based, QoS-enabled component middleware to support the development of safety-critical distributed systems. However, the approach lacks formal semantics.

The feature comparison of Gamma and the following tools can be found in Table 4: a SystemC modeling environment (SystemC ME) connected to the STATE tool, Æmilia ADL/TwoTowers, CHESS, Ptolemy II, BIP, UML-RT, AutoFOCUS 3, xtUML, COMDES-II and ProCom.

SystemC ME In [44], a modeling environment is introduced that supports the graphical definition of SystemC [45] models. SystemC is a C++ library offering classes and macros, which provide an event-driven simulation interface suitable for simulating concurrent processes. The basic building block of a SystemC model is the module, which represents computational parts of the design. Modules are composed of processes, ports, events, channels and variables. Processes, whose behavior can be defined using a state machine formalism, are the main computation elements of the module; they are concurrent and are used to describe functionality. Ports allow communication from the inside of a module to the outside on the basis of events declared by interfaces. Ports can be connected by channels. The environment supports the automatic SystemC code generation from the created models. The supported part of the SystemC language is given a formal semantics by connecting the modeling environment to the STATE tool [46]. STATE maps the informal SystemC code to a formal timed automaton formalism, UPPPAL, thus provides formal verification capabilities. Contrary to Gamma, this modeling environment currently does not support the hierarchical and mix-and-match composition of modules.

Æmilia ADL/TwoTowers Æmilia [47] is an architecture description language (ADL) based on EMPA$_{gr}$ process algebra, a compositional specification language of algebraic nature integrating process algebra theory and stochastic processes. This language supports the modeling of component-based software systems at the software architecture level. Designers have to start the modeling process by defining the behavior of the component types in the system and their interactions with the other components. Stochastic aspects, e.g., component interaction time, of the software architecture targeted for functional or extra-functional analysis (security and performance) have to be defined at this level. Next, instances of component types have to be defined along with their interactions in order to enable their communication. Based on the received composite models, integrated, functional and performance semantic models can be generated automatically, which can undergo formal analysis executed by the TwoTowers tool. The main difference between the Æmilia ADL and the languages of Gamma is that Gamma puts the focus on discrete state-based composition instead of stochastic process algebra and currently does not support the definition of

Table 4 Features of Gamma and its competitors

	Native statecharts	Mix-and-match composition	Formal semantics	Synchronization-based	Event-based	Stochastic proc. alg.-based	Formal verification	Model-based test generation	Code generation	Simulation
Gamma	✓	✓	✓		✓		✓	✓	✓	
SystemC ME			✓		✓		✓		✓	✓
Æmilia			✓			✓	✓			✓
CHESS	✓		✓		✓		✓		✓	
Ptolemy II		✓	✓	✓	✓		✓̸		✓̸	✓
BIP			✓	✓			✓		✓	✓
UML-RT	✓	✓̸	✓		✓		✓	✓	✓	✓
AutoFOCUS 3	✓	✓̸	✓		✓		✓	✓	✓	✓
xtUML	✓				✓				✓	✓
COMDES-II	✓		✓		✓		✓		✓	
ProCom	✓		✓		✓		✓		✓	

✓ = full support; ✓̸ = experimental

stochastic behavior.[24] Therefore, if stochastic processes are necessary to model the system, Æmilia has a clear advantage over Gamma. However, while process algebras generally focus on behavioral compositionality, Gamma prefers structural composition, which can be convenient and expressive, especially if the development process requires discrete states, mix-and-match composition and related code generation.

CHESS CHESS [48] is an open source methodology and toolset that aims to address safety, reliability, performance and other non-functional properties, while guaranteeing correctness of component development and composition. The CHESS methodology relies on the CHESS Component Model, which is built around the concept of components, containers and connectors. A component represents a purely functional unit, whereas the non-functional aspects are in charge of the infrastructure of the component and delegated to the container and connectors. The container is a wrapper that envelopes the component and is responsible for the realization of the non-functional properties. The connector is responsible for the interaction between components. Non-functional attributes are specified by annotating the interfaces of the component with non-functional properties, e.g., regarding real-time concerns, the activation pattern (such as sporadic or cyclic) can be specified for each provided operation of the component. CHESS models can be defined in the CHESS Modeling Language, which serves as an extension of the UML, SysML and MARTE modeling languages. Contrary to Gamma, CHESS does not focus on the mix-and-match composition of components.

BIP BIP [49] (Behavior, Interaction, Priority) is a modeling framework supporting the formal definition of heterogeneous systems. The BIP language supports the layered definition of hierarchical composite systems, defining three layers. The lowest layer specifies the behavior of system components, atomic or compound, using a variant of the Petri net formalism. The intermediate layer consists of a set of connectors linking ports together, thus defining the interactions between transitions of components. Contrary to Gamma, these interactions are based on synchronization. The top layer includes a set of dynamic priority rules between interactions and can be used for the specification of scheduling policies. BIP defines a clear operational semantics, which describes the behavior of both atomic and compound components. The behavior of atomic components is based on a rigorous transition system model; thus, formal verification of invariant properties and deadlock-freedom is also supported.

Ptolemy II Ptolemy II [50] is a modeling framework supporting the definition of hierarchical composite systems with diverse component types and interaction semantics. Modeling components, called actors in Ptolemy II, can be regarded as independent software modules. They are able to interact with each other by sending messages through interconnected ports. Models are created by composing actors, which is supported at multiple hierarchy levels. The interactions of actors can be executed with different semantic variations, defined by models of computation (MoC). Ptolemy II offers numerous MoCs that rigorously define the interaction between actors, e.g., process network, synchronous reactive and synchronous dataflow. The implementation of a MoC is called director. Each level of hierarchy in a model must have a single director that specifies the MoC. Directors of various hierarchy levels may have different types. Still, the composition of such heterogeneous components adheres to a rigorous semantics, which is a very powerful facility of Ptolemy II. Nevertheless, Ptolemy II offers only experimental formal verification capabilities [51].

UML-RT UML-RT [52] is a UML profile (evolved from the ROOM language [53]) used by IBM Rational Software Architect RealTime Edition (RSA RTE) and alternatively by Papyrus RT [54]. It facilitates the modular development of software systems. The language supports synchronous and asynchronous communication, hierarchy and also provides various action languages, such as Java or C++. The basic building block of a UML-RT model is a capsule, whose behavior can be defined using statecharts. Additionally, UML-RT models describe connections to other capsules with the help of structure diagrams. A capsule can contain parts, which are instances of other capsules, thus, hierarchical modeling is supported. Capsules and parts communicate through ports. Source code generation from UML-RT models is also supported [54]. UML-RT models can be transformed to a rigorous state machine formalism, called CFFSM [55,56], which can be formally analyzed by their model checker based on CTL expressions. Nevertheless, this tool does not fully support mix-and-match composition of components. Test generation is also supported in [57]. UML-RT models could be integrated with other models in the Gamma framework as UML-RT components and their composition can be expressed using the statechart language of Gamma.

AutoFOCUS 3 In [58], a model-based tool and research platform for safety-critical embedded systems is introduced. AutoFOCUS 3 supports the design, development and validation of safety-critical embedded systems in many development phases, including architecture design, implementation, hardware/software integration and safety argumentation based on formal models. The formal semantics of the approach is based on the FOCUS method defined in [59]. AutoFOCUS 3 (similarly to Gamma) employs a multi-level transformation chain to produce formal models from high level design models. Furthermore, it has a modeling language based on the statechart formalism and the tool provides editor support too. As opposed to AutoFOCUS 3, our Gamma-based approach promotes to use a common formal

[24] However, an extension to Gamma introducing stochastic behavior is realizable and in fact under development.

representation (GSL) and integrate the models of the different design tools: models developed in the AutoFOCUS 3 tool could also serve as input for Gamma. Consequently, Gamma could serve as an integration tool for AutoFOCUS models. AutoFOCUS 3 provides formal verification mainly for synchronously (according to weak or strong causality) composed software models [60] as it is integrated with well-known symbolic model checkers, such as NuSMV/nuXmv, which provide efficient verification capabilities for synchronous systems. On the contrary, Gamma uses UPPAAL for model checking, which is specialized to verify timed and distributed systems.

xtUML xtUML (eXecutable and Translatable UML) [61] is a UML-based modeling language with a special focus on executable semantics. State machines with an expressive action language are used to define the behavior of the components. BridgePoint xtUML is an xtUML design tool that supports code generation through model compilers and also provides simulation capabilities. xtUML is widely used in the industry; however, there is no formal verification support for xtUML models as far as we know.

COMDES-II COMDES-II (COMponent-based design of software for Distributed Embedded Systems - version II) [62] is a design tool for layered component models where synchronous and asynchronous behavior are explicitly separated into different layers. State machines and function block models (FBD) are used to define behavior while an actor-based architecture modeling language is used to represent the static aspects. The COMDES-II approach facilitates the development of real-time embedded systems by providing rigorous design and analysis methods. Verification of COMDES-II models [63] is based on the UPPAAL model checker.

ProCom ProCom [64] is a component model for real-time and embedded systems. ProCom employs a layered component model as it consists of two distinct, but related layers. At the upper layer (called ProSys), the system is modeled as a number of active and concurrent subsystems, which communicate by message passing. The lower layer (ProSave) addresses the internal design of a subsystem that can be implemented by code. ProCom has a formal semantics [65,66] that focuses on the reactive and real-time aspects of the systems and supports the co-existence of black-box and fully implemented components.

10 Conclusion

The Gamma framework is a modeling tool supporting the hierarchical design, implementation and verification of state-based reactive systems. Gamma has an extensive language family, integrates statechart components from the Yakindu Statechart Tools and MagicDraw, provides a Java code generator for implementation of composition-related code and applies the UPPAAL model checker for formal verification and test generation. The extensible architecture of the framework allows additional tools and features to be plugged in.

The main contribution of this work is the design and formalization of the Gamma Composition Language that supports the definition of interfaces, ports and bindings, enabling individual components to serve as endpoints. Communication is provided by channels connecting port instances. Relying on these elements, we have defined various kinds of composition modes for hierarchical composite model building. The three distinguished composition modes are the asynchronous-reactive, the synchronous-reactive and cascade.

Asynchronous components represent independently running components communicating with immutable messages stored in message queues. This semantics is suitable for designing separate units executed in their own processes. Synchronous-reactive components are useful for providing a single executing unit consisting of multiple, functionally independent components. This composition mode is beneficial for the design of low-level, tightly-coupled controllers. Cascade composition is practical for designing units with a pipeline-like behavior: the input given into the model is processed by multiple consecutive filters, where a single filter can be executed one or multiple times. We believe that these composition methods cover a large portion of the problems emerging in the design of reactive systems.

The precise semantics of the aforementioned composition modes allowed us to implement the code generation and verification functionalities of the Gamma framework. All elements of the GCL are supported during code generation and most during verification. Regarding test generation, test suites for state coverage and transition coverage can be generated.

As for future work, we plan to integrate ongoing side-projects to extend the Gamma framework with additional functionalities, e.g., the simulation of composite components. Also, we would like to introduce source code generation directly from Gamma statecharts as well as include additional programming languages, such as C/C++. Moreover, we are working on extending the framework with additional model checkers, such as Theta [67]. Regarding lessons learned in the case studies, we intend to provide support for handling symmetry in models (e.g., new element types, such as array and structure elements), complex ports that simplify the connection of components with many matching ports, as well as larger control over timing in the asynchronous composition semantics (similarly to [30]). Furthermore, in the next version of the framework we plan to develop methods for the verification of dynamic architectures, e.g., cyber-physical systems, by supporting dynamic, contract-based definition of components and connections.

The verification of such models could be based on both development-time and runtime verification methods.

By offering multiple modeling aspects, composition semantics, code generation and verification features in a single, extensible framework, we hope that Gamma can assist system and software engineers in leveraging the potential of model-driven development. As Gamma is now open-source, we also hope that the results of our research influence and aid fellow researchers and developers in developing their modeling tools.

Acknowledgements Open access funding provided by Budapest University of Technology and Economics (BME). This work was partially supported by Új Nemzeti Kiválóság Program 2017-2018, Nemzeti Tehetség Program, Nemzet Fiatal Tehetségeiért Ösztöndíj 2018 (NTP-NFTÖ-18) and by the ÚNKP-19-3 New National Excellence Program of the Ministry for Innovation and Technology.

References

1. Nuzzo, P., Sangiovanni-Vincentelli, A.L., Bresolin, D., Geretti, L., Villa, T.: A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. Proc. IEEE **103**(11), 2104–2132 (2015)
2. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**, 231–274 (1987)
3. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of UML statechart diagrams. In: Formal Methods for Open Object-Based Distributed Systems, Springer, pp. 331–347 (1999)
4. Crane, M. L., Dingel, J.: On the semantics of UML state machines: categorization and comparision. In: In Technical Report 2005-501, School of Computing, Queen's (2005)
5. Group, O. M.: Semantics of a foundational subset for executable UML models. Technical Report formal/2018-12-01, Object Management Group (2018)
6. Group, O. M.: Precise semantics of UML state machines (PSSM). Technical Report formal/2019-05-01, Object Management Group (2019)
7. Group, O. M.: Precise semantics of UML composite structures (PSCS). Technical Report formal/2019-02-01, Object Management Group (2019)
8. Crane, M. L., Dingel, J.: UML versus classical versus Rhapsody statecharts: not all models are created equal. In: Briand, L., Williams, C. (eds.), Model Driven Engineering Languages and Systems, Springer, Heidelberg, pp. 97–112 (2005)
9. Eshuis, R.: Reconciling statechart semantics. Sci. Comput Program. **74**(3), 65–99 (2009)
10. Molnár, V., Graics, B., Vörös, A., Majzik, I., Varró, D.: The Gamma Statechart Composition Framework. In: 40th International Conference on Software Engineering (ICSE 2018), ACM, Gothenburg, Sweden (2018)
11. Behrmann, G., David, A., Larsen, K. G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0 (2006)
12. Vörös, A., Búr, M., Ráth, I., Horváth, Á., Micskei, Z., Balogh, L., Hegyi, B., Horváth, B., Mázló, Z., Varró, D.: MoDeS3: model-based demonstrator for smart and safe cyber-physical systems. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.), NASA Formal Methods, Springer, Cham, pp. 460–467 (2018)
13. Lee, E.A., Parks, T.M.: Dataflow process networks. Proc. IEEE **83**, 773–801 (1995)
14. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. Proc. IEEE **79**(9), 1270–1282 (1991)
15. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)
16. Edwards, S.A., Lee, E.A.: The semantics and execution of a synchronous block-diagram language. Sci. Comput. Program. **48**(1), 21–42 (2003)
17. Clarke Jr., E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking. MIT press, Cambridge (2018)
18. Dwyer, M. B., Avrunin, G. S., Corbett, J. C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002), pp. 411–420 (1999)
19. Hegedüs, Á., Bergmann, G., Ráth, I., Varró, D.: Back-annotation of simulation traces with change-driven model transformations. Proc. Softw. Eng. Formal Methods (SEFM) **2010**, 145–155 (2010)
20. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**, 297–312 (2012)
21. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J. S.: A formal semantics for complete UML state machines with communications. In: Johnsen, E.B., Petre, L. (eds.), Integrated Formal Methods, Springer, Berlin, pp. 331–346 (2013)
22. Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools, pp. 87–124. Springer, Berlin (2004)
23. Wagner, F.: VFSM executable specification. In: CompEuro 1992 Proceedings Computer Systems and Software Engineering, pp. 226–231 (1992)
24. Graics, B., Molnár, V.: Documentation of the Gamma Statechart Composition Framework v0.9. Technical Report, Budapest University of Technology and Economics, Department of Measurement and Information Systems. https://inf.mit.bme.hu/en/gamma/ (2016)
25. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. **19**(3), 215–261 (2007)
26. Callahan, J., Schneider, F., Easterbrook, S.: Automated software testing using model-checking (2000)
27. Graics, B., Majzik, I.: Modeling and analysis of an industrial communication protocol in the Gamma framework. In: 27th Minisymposium, Department of Measurement and Information Systems, Budapest, Hungary (2020)
28. Procter, S., Feiler, P.: The AADL error library: an operationalized taxonomy of system errors. Ada Lett. **39**, 63–70 (2020)
29. Clarke, E. M., Emerson, E. A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logic of Programs, Springer, pp. 52–71 (1981)
30. Lohstroh, M., Schoeberl, M., Jan, M., Wang, E., Lee, E. A.: Programs with ironclad timing guarantees: work-in-progress. In: Proceedings of the International Conference on Embedded Software Companion, ACM, New York, NY, USA, October 13–18, 2019, p. 1 (2019)

31. Sztipanovits, J., Bapty, T., Neema, S., Howard, L., Jackson, E.: OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems, Springer, Berlin, pp. 235–248 (2014)
32. Simko, G., Lindecker, D., Levendovszky, T., Neema, S., Sztipanovits, J.: Specification of cyber-physical components with formal semantics—integration and composition. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.), Model-Driven Engineering Languages and Systems, Springer, Berlin, pp. 471–487 (2013)
33. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: Robochart: modelling and verification of the functional behaviour of robotic applications. Softw. Syst. Model. **18**(5), 3097–3149 (2019)
34. Jiang, Y., Song, H., Yang, Y., Liu, H., Gu, M., Guan, Y., Sun, J., Sha, L.: Dependable model-driven development of CPS: from stateflow simulation to verified implementation. ACM Trans. Cyber. Phys. Syst. **3**, 12:1–12:31 (2018)
35. Juodisius, P., Sarkar, A., Mukkamala, R.R., Antkiewicz, M., Czarnecki, K., Wasowski, A.: Clafer: lightweight modeling of structure, behaviour, and variability. Art Sci. Eng. Program. **3**(1), 1–2 (2018)
36. Lugou, F., Li, L. W., Apvrille, L., Ameur-Boulifa, R.: SysML models and model transformation for security. In: Conference on Model-Driven Engineering and Software Development (Modelsward'2016), Rome, Italy (2016)
37. Apvrille, L., Roudier, Y.: Model-Driven Engineering and Software Development, ch. Designing Safe and Secure Embedded and Cyber-Physical Systems with SysML-Sec, Springer International Publishing, Dordrecht, pp. 293–308 (2016)
38. Haber, A., Ringert, J. O., Rumpe, B.: Montiarc architectural modeling of interactive distributed and cyber-physical systems. arXiv:1409.6578 (2014)
39. Childs, A., Greenwald, J., Jung, G., Hoosier, M., Hatcliff, J.: Calm and Cadena: metamodeling for component-based product-line development. Computer **39**(2), 42–50 (2006)
40. Feiler, P. H., Gluch, D. P., Hudak, J. J.: The Architecture Analysis and Design Language (AADL): an introduction. Technical Report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst (2006)
41. Reisig, W.: Associative composition of reactive systems. Technical Report, Humboldt-Universität zu Berlin, Institut für Informatik (2017)
42. Broy, M.: Compositional refinement of interactive systems. J. ACM **44**, 850–891 (1997)
43. Gerard, S., Babau, J.-P., Champeau, J.: Model Driven Engineering for Distributed Real-Time Embedded Systems. Wiley-IEEE Press, New York (2010)
44. Chhokra, A., Abdelwahed, S., Dubey, A., Neema, S., Karsai, G.: From system modeling to formal verification. In: The 2015 Electronic System Level Synthesis Conference (2015)
45. Panda, P. R.: SystemC: a modeling platform supporting multiple design abstractions. In: Proceedings of the 14th International Symposium on Systems Synthesis, ACM, pp. 75–80 (2001)
46. Herber, P.: A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata, GmbH, Berlin (2010)
47. Bernardo, M., Donatiello, L., Ciancarini, P.: Stochastic process algebra: from an algebraic formalism to an architectural description language. In: IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation, Springer, pp. 236–260 (2002)
48. Mazzini, S., Favaro, J. M., Puri, S., Baracchi, L.: Chess: an open source methodology and toolset for the development of critical systems. In: EduSymp/OSS4MDE@MoDELS (2016)
49. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM) 2006, IEEE, pp. 3–12 (2006)
50. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y., Neuendorffer, S.: Taming heterogeneity: the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
51. Bae, K., lveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using real-time maude. Sci. Comput. Program. **77**(12), 1235–1271 (2012)
52. Selic, B.: Using UML for modeling complex real-time systems. In: Mueller, F., Bestavros, A. (eds.), Languages, Compilers, and Tools for Embedded Systems, Springer, Berlin, pp. 250–260 (1998)
53. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley, New York (1994)
54. Hili, N., Dingel, J., Beaulieu, A.: Modelling and code generation for real-time embedded systems with UML-RT and Papyrus-RT. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 509–510 (2017)
55. Zurowska, K.: Language specific analysis of state machine models of reactive systems. Ph. D. Thesis, Queen's Univerity, Canada (2014)
56. Zurowska, K., Dingel, J.: Language-specific model checking of UML-RT models. Softw. Syst. Model. **16**(2), 393–415 (2017)
57. Rapos, E. J., Dingel, J.: Incremental test case generation for UML-RT models using symbolic execution. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 962–963 (2012)
58. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: tooling concepts for seamless, model-based development of embedded systems. In: ACES-MB and WUCOR@ MoDELS, vol. 1508, pp. 19–26 (2015)
59. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, Berlin (2012)
60. Kanav, S., Aravantinos, V.: Modular transformation from AF3 to nuXmv. In: MODELS (Satellite Events), pp. 300–306 (2017)
61. Shlaer, S., Mellor, S.J.: Object-Oriented Systems Analysis: Modeling the World in Data. Yourdon Press, New York (1988)
62. Ke, X., Sierszecki, K., Angelov, C.: COMDES-II: a component-based framework for generative development of distributed real-time control systems. In: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), pp. 199–208 (2007)
63. Ke, X., Pettersson, P., Sierszecki, K., Angelov, C.: Verification of COMDES-II systems using UPPAAL with model transformation. In: 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 153–160 (2008)
64. Bureš, T., Carlson, J., Crnkovic, I., Sentilles, S., Vulgarakis, A.: ProCom: the progress component model reference manual. Mälardalen University, Västerås, Sweden (2008)
65. Vulgarakis, A., Suryadevara, J., Carlson, J., Seceleanu, C., Pettersson, P.: Formal semantics of the ProCom real-time component model. In: 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, pp. 478–485 (2009)
66. Borde, E., Carlson, J.: Towards verified synthesis of ProCom, a component model for real-time embedded systems. In: Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering, pp. 129–138 (2011)
67. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: a framework for abstraction refinement-based model checking. In: Stewart, D., Weissenbacher, G. (eds.), Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design

Bence Graics is a Ph.D. student at the Budapest University of Technology and Economics under the supervision of Dr. István Majzik and researcher at the MTA-BME Lendület Cyber-Physical Systems Research Group. He joined the Fault Tolerant Systems Research Group (ftsrg) during his undergraduate studies at BME and has been the main developer of the Gamma Statechart Composition Framework since the start of the project in 2016. His main research interest lies in the model-driven design and analysis of reactive systems using formal methods.

Vince Molnár is assistant professor at the Budapest University of Technology and Economics and researcher at the MTA-BME Lendület Cyber-Physical Systems Research Group. His main research field is model-based development and formal methods, with the primary focus on concurrent, distributed and safety-critical systems. He is the leader of the development of the Gamma Statechart Composition Framework and contributed to the development of PetriDotNet as well.

András Vörös is an assistant professor at the Budapest University of Technology and Economics and researcher at the MTA-BME Lendület Cyber-Physical Systems Research Group. His main research field is model-based development and verification of cyber-physical system. He has been involved in various projects, such as the PetriDotNet modelling and verification framework, the Gamma Statechart Composition Framework and the Theta configurable verification framework.

István Majzik is associate professor at the Budapest University of Technology and Economics. His fields of research and education include construction, evaluation and verification of dependable and safety-critical computer systems, and he is a co-author of more than 90 scientific papers. He is regular program committee member of international conferences in the field, among others he was general co-chair of the 35th IEEE Symposium on Reliable Distributed Systems (SRDS). He participated in several European research projects in the area of dependable embedded systems, model-based testing and formal verification.

Dániel Varró is a full professor of software engineering at McGill University and at Budapest University of Technology and Economics. He is also a research chair of the MTA Lendület Cyber-Physical Systems Research Group. He is a co-author of more than 150 scientific papers with seven Distinguished Paper Awards, and two Most Influential Paper Award. He regularly serves on the program committee of various international conferences in the field and serves on the editorial board of the Software and Systems Modeling journal (Springer) and Journal of Object Technology. He served as a program committee co-chair of FASE 2013, ICMT 2014, SLE 2016 and MODELS 2021 conferences. He delivered a keynote talk at the IEEE CSMR 2012 and the SOFSEM 2016 conferences and at various international workshops and at the DSM-TP international summer school. He is a co-founder of the VIATRA model query and transformation framework, and IncQuery Labs Ltd., a technology-intensive Hungarian company.