Chapter 2

Preliminaries

This chapter presents the theoretical foundations of the dissertation necessary to understand the subsequent chapters. The chapter starts with an overview of the well-known state machine formalism (see Section 2.1) serving as a theoretical basis in this work for defining and integrating state-based components. Next, Section 2.2 introduces statecharts, the extension of the state machine formalism, to lay the foundations for the high-level description of state-based behavior in the Gamma framework. Section 2.3 presents two running examples used throughout this dissertation. Building on the running examples, Section 2.4 overviews a low-level analysis formalism, called EXtended Symbolic Transitions Systems (XSTS), which serves as an intermediate representation for reactive behavior and its verification in the framework. Finally, Section 2.5 informally presents the composition modes that we formalize and realize in our framework's composition language.

2.1 State machines

Event-driven behavior is commonly defined on the basis of state machines. This section formally introduces the theory of *state machines* – an abstract representation of reactive behavior that different modeling aspects of this work builds on, including (1) the high-level statechart language for capturing standalone component behavior (an extension of the state machine formalism) and (2) the composition language that considers an abstract state machine based view (abstract behavioral contract) of components (see following paragraphs) during component integration.

State machines are a mathematical model of computation to describe the behavior of a reactive system, component or object in an event-driven way [BC14]. Formally, a deterministic, fully specified finite state machine is a 5-tuple: $M = (S, s^0, I, O, T)$ where:

- $S = \{s_1, s_2, \cdots, s_n\}$ is a finite set of *states*, i.e., stable situations of the state machine with $s^0 \in S$ being the initial state.
- I is a finite set of *input events* that are stimuli from the environment and O is a finite set of *output events* that are stimuli for the environment such that $I \cap O = \emptyset$.
- $T\colon (I \times S) \to (S \times O)$ is the fully defined *transition function* that represents changes of states in response to input events and generating output events meanwhile.

The *behavior* of a state machine, that is, the *operational semantics* of the formalism, is defined by the maximal set of *execution traces*. An execution trace consists of a sequence of *steps*. A step describes a change of state with an output event raised by the state machine in response to an input event. Formally, an execution trace of a state machine can be described as follows:

- $\rho = (i_0, s_0, s'_0, o_0), \cdots, (i_n, s_n, s'_n, o_n)$ is an execution trace, which consists of one or more steps: $n \in \mathbb{N}$.
- (i_i, s_i, s'_i, o_i) is a step, consisting of an input event $i_i \in I$, a source state $s_i \in S$, a target state $s'_i \in S$ and an output event $o_i \in O$. A step is considered valid if $T(i, s_i) = (s'_i, o_i)$.
- An execution trace ρ is considered valid if:
  - $s_0 = s^0$, that is, the source state of the first step is the initial state;
  - For each step (i_i, s_i, s'_i, o_i), $s_i = s'_{i-1}$, that is, the source state of a particular step is the target state of the previous step.

A state s of the state machine is reachable if an execution trace $\rho = (i_0, s_0, s'_0, o_0), \cdots, (i_n, s_n, s'_n, o_n)$ exists with $s = s'_n$ for some n.

2.2 Statecharts

There are various extensions to the state machine formalism that facilitate the compact modeling of hierarchical and concurrent systems [HU90]. The most relevant one is statecharts [Har87], which extends the state machine formalism's concept of *state* and *transition* with

- *auxiliary variables*, i.e., the valuations of variables are included in the states of the state machine;
- *actions* related to transitions and the entry/exit of states, which support variable handling on the basis of imperative code constructs (e.g., if-else constructs, for loops and function calls);
- (potentially hierarchical) *state refinement*, i.e., states can contain one or more (orthogonal) *regions* (composite states), each of which may contain additional states - during execution, an active region shall have a single active state;
- *composite transitions* to synchronize orthogonal regions or describe different choices during state changes; and
- *memory* to store the last active states (history) in a region.

Abstract syntax Figure 2.1 depicts an excerpt of the modeling elements and their possible interrelations (metamodel excerpt disregarding different types of triggers, expressions and pseudo states) used to define statechart components in this work (see Section 3.2 for the statechart language of the Gamma framework).

A statechart can contain *variable declarations* to store values of different types (e.g., boolean, integer and enumeration) and *timeout declarations* to capture time lapse and generate events that may trigger changes in the model.

Regarding hierarchy-related constructs, a statechart can contain one or more (orthogonal) top-level *regions* (top regions) as root elements for "static" *state node* constructs in the model. State nodes can either be *states* that represent stable "situations" in the model, or *pseudo states*, e.g., *fork*, *join*, *merge* and *choice states* or different *entry nodes*, which can be used to (de)compose *transitions* during state changes. States may execute certain *actions* upon entering (entry action) or leaving (exit action) the particular state, e.g., setting a timeout. In order to support the hierarchical decomposition of operation, *composite states* can contain one or more (orthogonal) regions that can contain additional state nodes. Each region has a single *entry node*, which defines the active state upon activation: in the case of an (1) *initial state*, the single outgoing transition determines the active state; in addition, regions may have (2) *shallow* or (3) *deep history states* that store/restore the last active state configuration upon deactivation/activation – shallow history states consider states only in the contained region, whereas deep history also considers the internal regions of contained composite states.

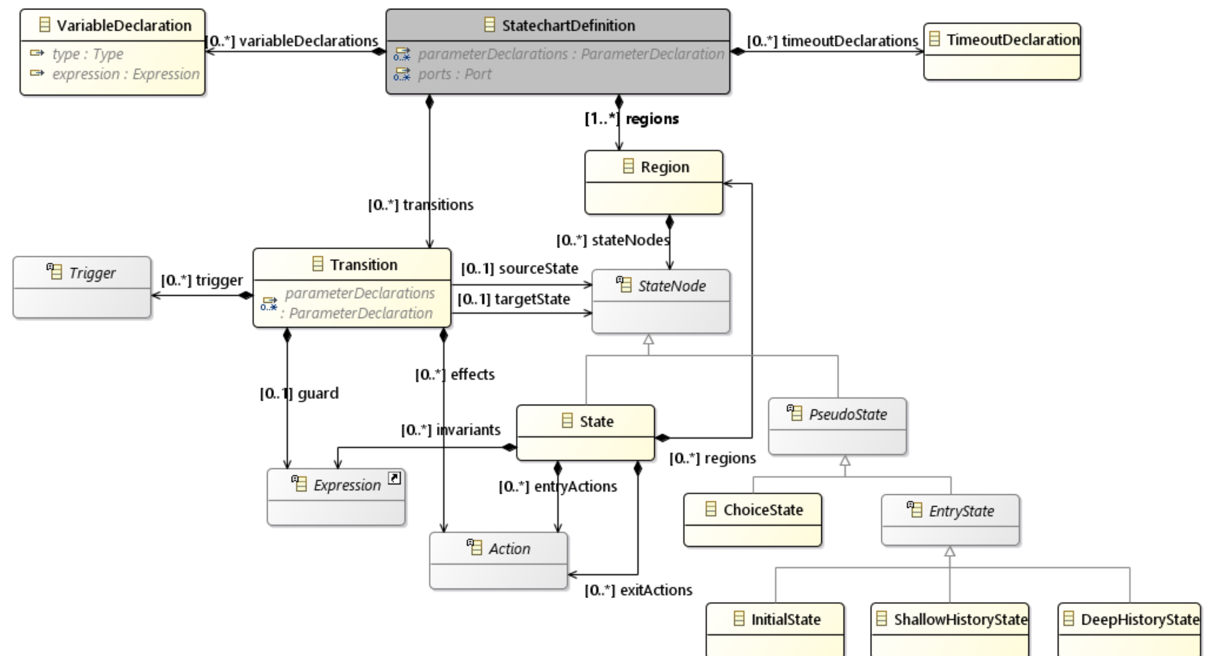

Figure 2.1: Elements of statechart components used in this dissertation (statechart language metamodel excerpt).

As for elements describing dynamic changes, *events* can be received via well-defined *ports* and associated to *transitions* as *triggers*, which describe atomic changes between stable states in the model. The enabledness of transitions (in addition to triggering events) can be controlled by *guard expressions* (boolean expressions); the effects of transitions can comprise event raisings, as well as variable and timeout handling and potentially additional constructs, e.g., branching and function calls, in accordance with the featured action language.

Complex transitions can be defined using pseudo states: *join* and *fork states* are used to synchronize between orthogonal regions (simultaneous deactivation and activation of their states, respectively), *merge states* syntactically merge incoming transitions with a single outgoing transition, whereas *choice states* describe possible outcomes (distinct choices) based on boolean (guard) expressions.

Concrete syntax Statechart models are generally represented graphically, even though some languages support their textual representation (e.g., the statechart language of Gamma, presented in Section 3.2). In the following, we present the generally used graphical syntax of statecharts.

Event, *parameter*, *timeout* and *variable declarations* are represented using a textual syntax, declaring their names (identifiers) and types.

Regions (both top regions of statecharts and regions contained by states) are graphically represented by coherent areas with dashed lines as delimiters. *States* are represented by rounded rectangles, their *entry* and *exit actions* are defined using a textual syntax according to the supported action language inside the corresponding rectangle after the *entry* and *exit* keywords and a slash symbol, respectively. *Initial states* are represented by black circles, whereas *history states* are represented by a circle with a *H* or *H** label inside (*shallow history* and *deep history*, respectively).

Transitions are represented by arrows, connecting the source and target state nodes, with labels representing the *trigger* (name of the triggering event), the *guard* expression inside square brackets and *effects* (statements according to the featured action language) after a slash symbol.

Regarding complex transition related pseudo nodes, *fork* and *join states* are represented as rectangles, *merge states* are represented as circles and *choices states* are represented as diamonds.

Semantics Subsequent sections (see Sections 3.2 and 3.4) propose contributions closely related to the semantics of statecharts. Therefore, in the following, we overview the operational semantics of UML/SysML statecharts in a structured, semi-formal way, which can be considered as the de facto standard for statechart semantics. Note that there exist several formalizations of the operational semantics of statecharts, e.g., [Czi+17; Pin07]; accordingly, giving a formal operational semantics to our framework's statechart language is out of the scope of this work. Instead, we aim at giving a denotational semantics for statecharts by means of *model transformations* in Section 3.4, which build on the semantics-related rules presented below and, as a novelty, also extend them by proposing customizable *semantic variation points* (denoted by SVP-* below to facilitate their later identification) to various model elements. The model transformations build on the XSTS formalism, necessitating the introduction of the extensions in the formalism to support high-level statechart models. In particular, the model transformations allow for the description of statechart behavior on the basis of XSTS models, and this way, capture the low-level *state machine* behavior (steps and execution traces, see Section 2.1) via the STS mapping (see Section 2.4).

In order to describe the operational semantics of statecharts, in addition to the *state machine* whose behavior is described by the statechart, we consider a "*runtime environment*" that comprises an *event queue* that stores incoming events, and a *scheduler* that retrieves events from the queue and passes them to the state machine. In general, the operational semantics defines the behavior of the state machine when it *processes* an *event*, i.e., a single *step* of the state machine that comprises the *firing* of enabled *transitions*. The basic properties of the semantics are as follows:

- Events are processed *one by one*, i.e., the scheduler passes a new event only if the previous event has been completely processed.
- A *complete* processing of events is conducted (run to completion): a maximal set of (nonconflicting) transitions fires, i.e., every enabled transition fires unless prevented by a conflict. The next event is passed only after every firing has completed.

Based on the above properties, the four phases of event processing are as follows.

Phase 1 In the first phase, the scheduler retrieves an event from the event queue and passes it to the state machine, which is followed by the evaluation of the transitions' *enabledness* of the state machine: a transition is *enabled*, if

- its source state is active,
- the passed event triggers the transition, and
- the guard condition evaluates to true (SVP-1).

Based on the number of enabled transitions, the event processing can continue in one of the following ways: in case (1) *no* transition is enabled, then the step finishes and a new event is processed; (2) *one* transition is enabled, then this transition is fired (see Phase 3); (3) *multiple* transitions are enabled, a set of firing transitions has to be selected based on *conflict* and *conflict-resolution* (see Phase 2); after that, a maximal set of nonconflicting transitions fires.

Phase 2 Two transitions are *conflicting* if the sets of states (including the child states of composite states) that they leave are *not disjoint*. Conflicts are resolved based on *priority*: the priority of a transition is higher if its source state is *lower* in the refinement hierarchy (SVP-2). At the same hierarchy level (i.e., for states leaving the same state), conflict-resolution is conducted nondeterministically, i.e., one transition is selected (SVP-3).

Phase 3 Next, the selected (nonconflicting) transitions fire in a nondeterministic order (SVP-4). Note that, as a result, the order of the action's execution is also nondeterministic. A single transition is fired in the following way:

1. The source states are exited, starting from the lower hierarchies first, while executing their exit actions.
2. The effects (actions) of the transition are executed.
3. The target states are entered, starting from the higher hierarchies first, while executing their entry actions.

Phase 4 As a last step, a new stable state configuration is entered according to the target state(s) of the firing transition(s):

- If the target state is *simple* (not composite), then it will be part of the new configuration (will be activated), while its parent states (in which it is a child state) will also be activated. Activated parent states will activate a child state in each of their regions in a nondeterministic order (SVP-4) determined by the initial state.
- If the target state has a *single region*, then a single child state will be activated.
- If the target state has *multiple orthogonal regions*, then a single child state will be activated in each region in a nondeterministic order (SVP-4 – same SVP as above as both rules target orthogonal regions).
- If the target is a *history state*, then the most recent state configuration is restored considering only a single hierarchy level (shallow history) or also every contained region (deep history). At the first activation of the region, the default state is activated.
- If the target state is a *pseudo state*, then one of its outgoing transitions is fired by iterating Phases 1-4.

2.3 Running examples

The following paragraphs introduce a model from the aerospace domain (*simple space mission*), a simplified *elevator system* and a *railway interlocking subsystem* which we use as running examples in the rest of the dissertation to demonstrate the applicability of our framework's modeling languages and functionalities in the context of composite state-based models.

Simple space mission The *simple space mission* model was originally proposed by NASA based on SysML in the context of the OpenMBEE[1] framework. Previously, we used the model in the context of a case study in [*j*1] where we investigated how SysML models comprising hierarchical state machines and activity diagrams can be mapped into the statechart and composition languages of the Gamma framework to capture communication between a *spacecraft* and a *ground station* component.[2] The

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

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

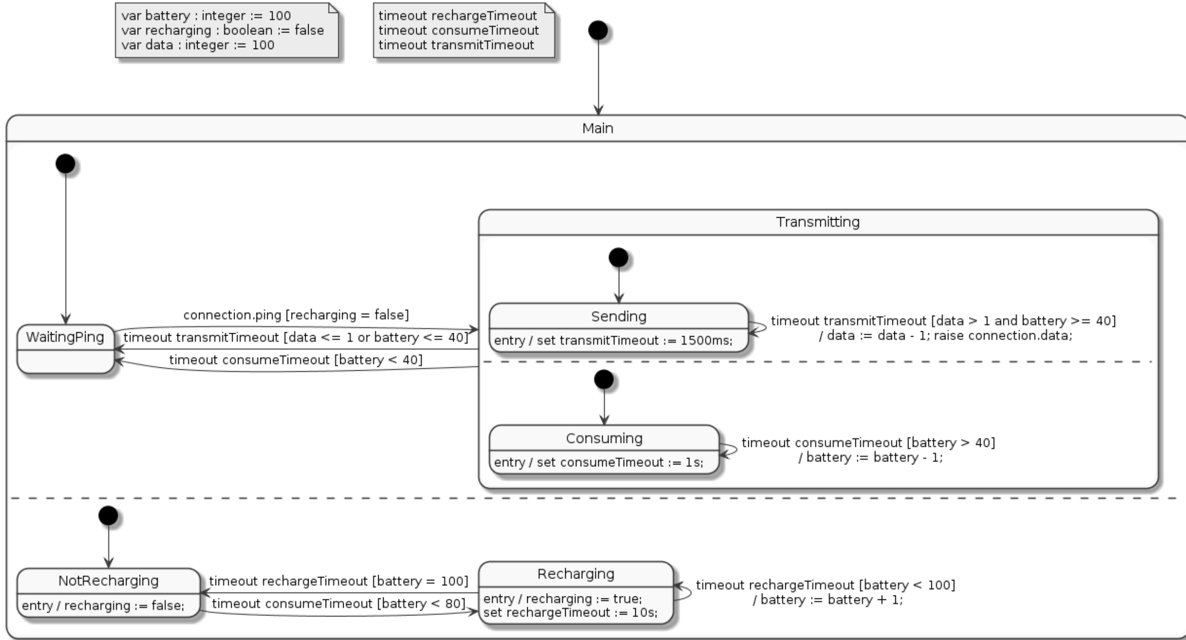

Figure 2.2: *Spacecraft* component of the *simple space mission.*

emergent *spacecraft* and *ground station* statechart models are visualized in Figures 2.2 and 2.3, respectively. Figure 2.4 shows an abstract version of the *spacecraft* component to allow for the simpler and more concise demonstration of modeling elements and features in subsequent sections.

The modeled system operates as follows. The *ground station* receives control events from its environment (*start* and *shutdown*) via its *control* port, and can ping the *spacecraft* (*ping* event) to initiate incoming data transmission. The component has several timeouts to handle the absence of incoming events. The *spacecraft* starts the data transmission upon the reception of a *ping* event, transmitting

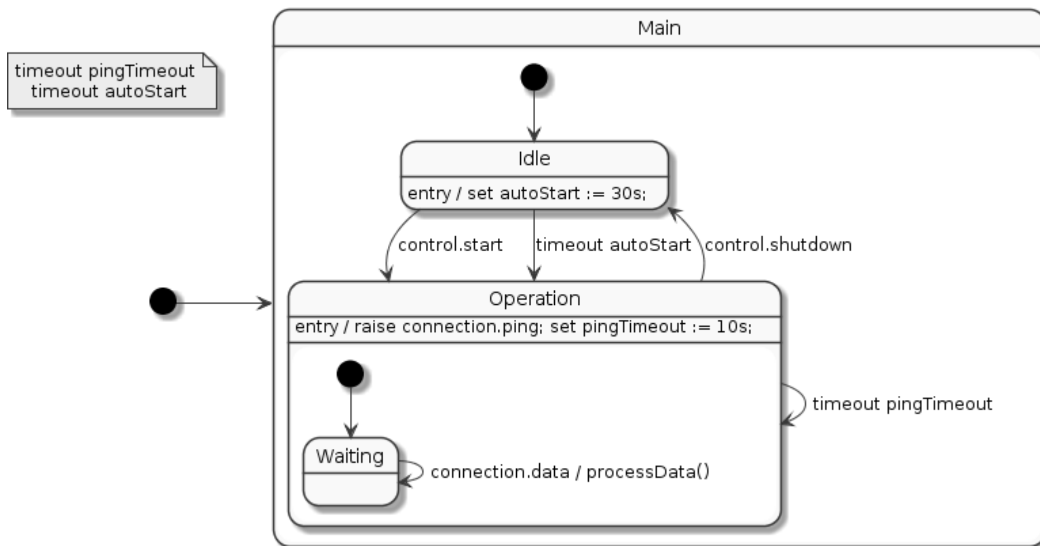

Figure 2.3: *Ground station* component of the *simple space mission.*

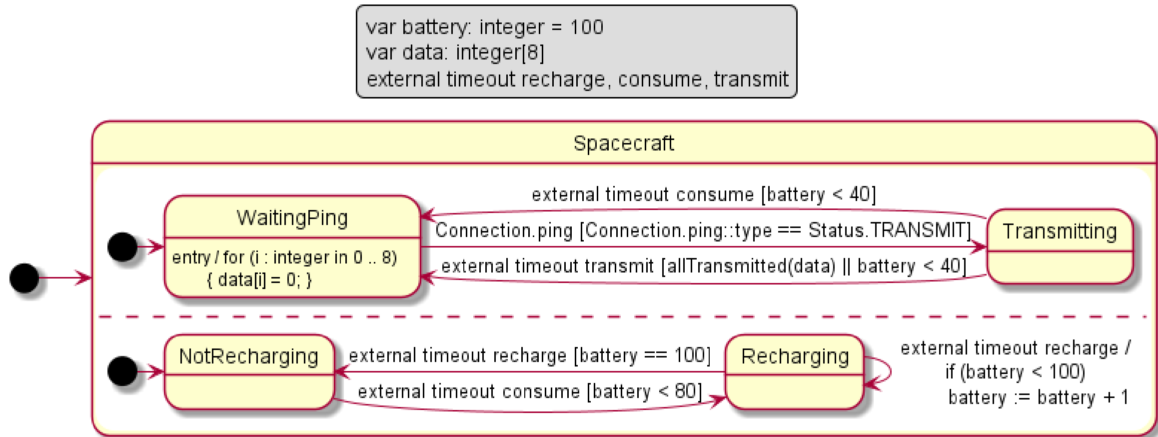

Figure 2.4: Abstract version of the *spacecraft* component model in the *simple space mission*.

data in packets via the *connection* port (variable *data* stores the number of remaining packets). Data transmission for the *spacecraft* requires energy, denoted by the *battery* variable. If the battery goes too low, the *spacecraft* enters a recharging state where energy is restored. Similarly to the *ground station*, the *spacecraft* has timeouts to measure time lapse and handle idleness.

Elevator system The simplified *elevator system* consists of two state-based components first presented in [SP10]. The components of the system are modeled in Yakindu (see Figure 2.5), comprising a *cabin controller* that, in addition to an idle state, can initiate the movement of the elevator cabin up and down in accordance with external commands (*Cabin.up* and *Cabin.down* events) and control the *cabin door controller* to open or close (*Door.open* and *Door.close* events) the cabin door.

Railway interlocking subsystem The *railway interlocking subsystem* (RIS) model (depicted in Figure 2.6) altogether consists of 38 states, 118 transitions and 23 variables (including timeout declara-

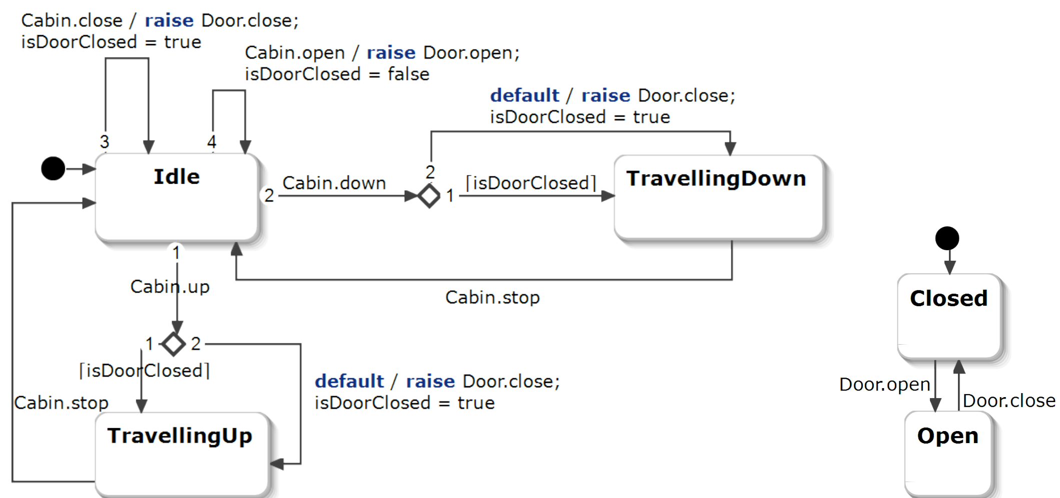

Figure 2.5: *Cabin controller* and *cabin door controller* models of the *elevator system*.

Figure 2.6: Railway interlocking subystem (RIS) topology.

tions) and represents the realization of an industrial communication protocol (called Rigel) used in interlocking systems. It comprises three components defined in the proprietary XSML language (integrated [e16] to the Gamma framework), namely *controlCenter*, *dispatcher* and *objectHandler*. The components are executed sequentially, and communicate with their messages stored in message queues; the *controlCenter* and *objectHandler* can communicate only via the *dispatcher*. For additional details, we direct the reader to [e16].

2.4 EXtended Symbolic Transition Systems

The *symbolic transition systems* (STS) formalism [HM00] is a commonly used low-level representation, e.g., for hardware model checking. As a main feature, STS models consist of two Satisfiability Modulo Theories (SMT) [BT18] formulas that describe the *set* of *initial states* and the *transition relation*. As STS serves as a foundation for our intermediate formalism to capture reactive behavior in the Gamma framework, we present its formal definition in the following paragraphs.

Symbolic transition system A *symbolic transition system* is a tuple $STS = (V, T, I)$ where:

- $V = \{v_1, \cdots, v_n\}$ is the set of *variables* with *domains* $D_{v_1}, \cdots, D_{v_n}$.
- T is the *transition formula* over $V \cup V'$ that describes the *transition relation* between the values of variables corresponding to the *current state* $v \in V$ and the *next state* $v' \in V'$. The result of the iterative application of T on V is denoted by $V^{(i)}$ where i denotes the number of iterations. Accordingly, $V = V^{(0)}$ and $V' = V^{(1)}$.
- I is the *initial state formula* over V that describes the values of the variables in the initial states.

A *concrete state* $s \in S \subseteq D_{v_1} \times \cdots \times D_{v_n}$ is an interpretation that assigns a value $s(v) \in D_v$ to each variable $v \in V$ of its domain D_v. A *concrete state* can also be regarded as a tuple of values $s(v_1), \cdots, s(v_n)$. A state with an index $s^{(i)}$ assigns values to the elements of $V^{(i)}$. Given an SMT formula ϕ let $s \models \phi$ denote that assigning the variables in ϕ with the values in s evaluates to true.

The set of *initial states* is $\{s \mid s \models I\}$. A *transition* exists between two states s and s' if $(s, s') \models T$.

The *behavior* of a symbolic transition system, that is, the *operational semantics* of the formalism, is defined by the maximal set of *concrete paths*. A *concrete path* is a finite sequence of concrete states $\sigma = s_0, s_1, \cdots, s_n$, for which $\{s_0 \models I\}$ and $(s_1^{(1)}, \cdots, s_n^{(n)}) \models \bigwedge_{0 \leq i < n} T^{(i)}$, where $T^{(i)}$ denotes the *transition formula* over $V^{(i)} \cup V^{(i+1)}$ i.e., the path starts in an initial state, and the successor states satisfy the transition relation. A concrete state s is reachable if a path $\sigma = s_1, \cdots, s_n$ exists with $s = s_n$ for some n.

Interrelations with state machines Note that the STS formalism is expressive enough to capture behaviors modeled by a state machine:

- The state machine's set of states S is represented by a v_s variable (corresponding to the active state during execution); $S = D_{v_s} = \{l_{s_1}, \cdots, l_{s_n}\}$ where s_i is represented by literal l_{s_i}; accordingly, initial state s_0 is represented by l_{s_0}.

- The state machine's set of input and output events I and O are represented by variable sets V_I and V_O, respectively, such that $V_I \cap V_O = \emptyset$ and $\forall v \in V_I \cup V_O : D_v = \{\bot, \top\}$, i.e., each input i and output event o is represented by boolean variables v_i and v_o, respectively.
- The state machine's fully defined *transition function* T is represented by the *transition formula* $(V_I \cup v_s) \cup (v'_s \cup V'_O)$ that describes the *transition relation* between the values of variables corresponding to the *input event* $v_i \in V_I$ and source state v_s, as well as the target state v'_s and output event $v'_o \in V'_O$.

Regarding execution, a step (i, s, s', o) of an execution trace is represented by an SMT formula of the concrete path $(v_i \wedge v_s = l_s \wedge \forall v \in (V_I \setminus \{v_i\}) : \neg v) \wedge (v'_{s'} = l_{s'} \wedge v'_o \wedge \forall v' \in (V_O \setminus \{v'_o\}) : \neg v')$. The initial state of the state machine is represented by the I SMT formula $v_s = l_{s_0} \wedge \forall v \in V_I \cup V_O : \neg v$.

EXtended Symbolic Transition Systems The *EXtended Symbolic Transition Systems*[3] (XSTS) is intended to serve as a *formal intermediate representation* for verifying reactive systems. The language offers an *imperative layer* (set of control structures) above the SMT formulas of STS models, enabling (1) the *efficient transformation* of *high-level design models* into XSTS, (2) the *mapping* of XSTS into the *input languages* of various *model checker back-ends*, such as UPPAAL or Spin, while (3) being *expressible* as SMT formulas using STS constructs and thus, verifiable by SMT-based model checkers such as Theta. The mapping of XSTS elements into STS formulas is described in Appendix A.

We introduce the elements of the XSTS language in Figure 2.7 based on the abstract example *spacecraft* statechart model depicted in Figure 2.4. Note that the example XSTS model, besides the behavior of the spacecraft component that features *message queue* based communication, also includes the explicit description of its *environment* in a separate transition, i.e., the handling of variables corresponding to *input events* and their parameters before considering the model behavior (see **Transitions** paragraph below).

Type declarations The textual representation of an XSTS model (see Figure 2.7) begins with *custom type* declarations (**type** keyword), which are similar to enum types in programming languages. For instance, the literals of a type declaration can be used to represent the states of a region in a statechart.

Variable declarations Type declarations are followed by global *variable declarations* (**var** keyword) with *integer*, *boolean*, and the previously discussed *custom* types (see the set of variables V in the STS definition). The language also supports *array* types, which are mathematical SMT arrays, similar to the *map* data structure of programming languages (see Line 12). Variable declarations can optionally contain an *initial value* (Line 8). Variables annotated with the **ctrl** keyword (Lines 6-7) are *control variables*, indicating that these variables contain *control* information, which can be exploited during verification, e.g., in Theta when using different abstraction algorithms.

Transitions Model behavior and the behavior of the environment (needed to allow for formal verification) are defined by three *transitions*. The **init** transition (Lines 48-50) can be used to initialize the model (*initial state formula I* of an STS). The model's internal behavior is described by the **trans** transition (Lines 14-47), while the behavior of the model's environment is described by the **env** transition (Lines 51-61); these transitions can be considered as the sequential decomposition of the *transition formula T* of an STS. In our example, the **init** transition sets the component's initial state, while the **env** transition places a random event (more precisely, its identifier) in its message queue, which is popped by the **trans** transition and processed in both regions of the component.

[3] The formalism is the result of joint work with Vince Molnár and Milán Mondok.

```
type Status : { TRANSMIT, ... }
type Communication : { _Inactive_,  WaitingPing, Transmitting }
type Battery : { _Inactive_, Recharging, NotRecharging }
var ping : boolean = false
var ping_types : Status = TRANSMIT
ctrl var _communication : Communication = _Inactive_
ctrl var _battery : Battery = _Inactive_
var battery : integer = 0
var recharge, transmit, consume : clock
// Message queue related variables
var size : integer
var eventQueue : [integer] -> integer
var argumentQueue : [integer] -> Status
trans {
 if (size > 0) {
  local var id : integer := eventQueue[0];
  eventQueue[0] := 0; // Event pop
  if (id == 1) {
   ping := true;
   ping_types := argumentQueue[0];
   argumentQueue[0] := 0; // Argument pop
  } ... // Potentially other eventIds
  size := size - 1; ... // Output event clearing
  par {
   if (_communication == WaitingPing && ping && ping_types == TRANSMIT) {
    _communication := Transmitting;
   } else if (...) {...} // Second transition
   else if (_communication == Transmitting && 1500 <= transmit) {
    _communication := WaitingPing;
    for i from 0 to 7 do { data[i] := 0; }
   }
  } and {
   choice {
    assume (_battery == NotRecharging && 1000 <= consume && battery < 80);
    _battery := Recharging;
    recharge := 0;
   } or {... // Second transition
    assume (_battery == Recharging && 1000 <= recharge && battery < 100);
    battery := battery + 1;
   } or {
    assume (_battery == Recharging && 10000 <= recharge && battery == 100);
    _battery := NotRecharging;
   } or {
    assume !(...); // Else branch
   }
  } ... // Input event clearing
} }
init { ... // Variable initializations
 _communication := WaitingPing; // Initial sates
 _battery := NotRecharging; ... } // Entry actions
env {
 if (size <= 0) {
  local var eventId : integer;
  havoc eventId;
  if (0 < eventId && eventId <= 1) {
   eventQueue[size] := eventId;
   local var argument : Status;
   havoc argument;
   argumentQueue[size] := argument;
   size := size + 1;
} } }
```

Figure 2.7: XSTS representation of the *spacecraft* component of Figure 2.4 with additional elements capturing message queue based communication and its nondeterministic environment.

Basic statements The detailed behavior of transitions is captured via *statements. Assign* statements (see Line 19) assign a value of its domain to a single variable. *Assume* statements (Line 34) act as guards; they can be executed only if their condition holds. *Havoc* statements assign a nondeterministically selected value of its domain to a variable (Line 54). *Local variable declarations* can be used to create *transient* variables that are accessible only in the scope they were created in and are not part of the model's state vector (Line 53).

Composite statements Composite statements contain other statements (operands), and can be used to describe complex control structures. *Sequences* are lists of statements that are executed sequentially; each statement of the sequence operates on the result of the previous statement. *Choice* statements (see Line 33) model nondeterministic choices between multiple statements; only one branch is selected for execution, which cannot contain failing assumptions, i.e., if every branch contains failing assumptions, then the choice statement also fails. *Parallel* statements support the parallel execution of the operands (see Lines 24-46). *If-else* statements are deterministic choices based on a condition (see Lines 25-31) with an optional *else* branch. The language also supports deterministic *for loops* over ranges (see Line 30).

As syntactic sugar, the language supports *unordered* and *orthogonal* statements (*unord* and *ort* keywords, respectively). *Unordered* statements define the execution of operands in a nondeterministic way; note that such statements can be mapped into choice statements with sequences as operands, containing the operands of the unordered statement in every possible permutation. In turn, *orthogonal* statements define the "independent" execution of their operands, i.e., when the potential effects (assign and havoc statements) of an operand *do not affect* the execution of the sibling operands. Such a behavior procures a constraint on the operands: they shall not write the same (global) variable. Note that orthogonal statements can be mapped into (1) a set of local variables that hold the values of the variables read by the operands (initialized at the beginning of the orthogonal statement) and (2) a *sequence* of the contained operands while changing the variable read references to the corresponding created local variables.

Semantics The *behavior* of an XSTS model, that is, the *semantics* of the formalism, is defined by mapping its structures into the STS formalism (see Appendix A for the mapping of statements). Regarding transitions, the **init** transition corresponds to the initial state formula I of STS, whereas the sequential composition of the **env** and **trans** transitions corresponds to the T transition formula of STS. Accordingly, regarding their execution order, the **init** transition is executed first (entering the initial state), after which the **env** and **trans** transitions alternate (generating successor states). The transitions are *atomic* in the sense that they are either *executed* in their *entirety* or *not at all* (recall the semantics of *assume* statements).

2.5 Composite reactive modeling

The systematic integration of state-based components (see Section 2.1) requires precise composition modes, which define the characteristics of their execution and interaction. Section 2.5.1 outlines the features of the *asynchronous-reactive* and *scheduled asynchronous-reactive* modes for asynchronous systems, and Section 2.5.2 overviews the *synchronous-reactive* and *cascade* modes for synchronous systems, which we support in our framework's composition language. Accordingly, the syntax and precise semantics of the language is presented in Section 3.3. The characteristics of the composition modes are illustrated graphically in Figures 2.8 and 2.9.

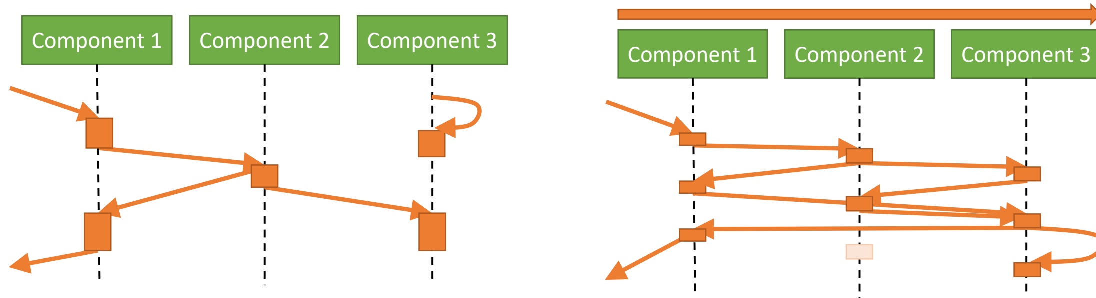

Figure 2.8: Sequence diagrams illustrating the execution and communication characteristics of the *asynchronous-reactive* and *scheduled asynchronous-reactive* composition modes, respectively.

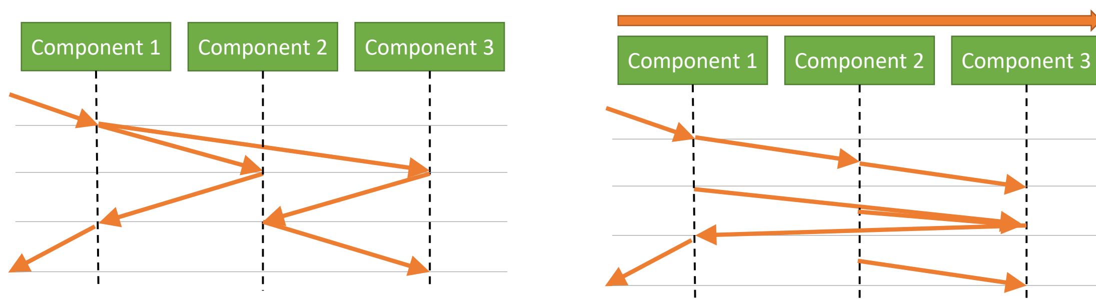

Figure 2.9: Sequence diagrams illustrating the execution and communication characteristics of the *synchronous-reactive* and *cascade* composition modes, respectively.

2.5.1 Asynchronous systems

In asynchronous systems [LP95], components represent *concurrent entities* that communicate with each other using *message queues*. Writing to a queue succeeds instantly, whereas reading from an empty queue blocks the reader (nonblocking-write, blocking-read approach). Message delivery is assumed to be reliable and thus, 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.[4] A read operation always retrieves a single message from the queue. As an 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 mode describes continuously and parallelly running components that react to events (messages) independently. This way, the execution frequency of system components is nondeterministic, but may be restricted with timing constraints. As an 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.

Scheduled asynchronous-reactive The scheduled asynchronous-reactive mode is the variant of the asynchronous-reactive mode that is restricted in terms of component execution: components are

[4]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.

executed *sequentially*, but they still communicate with immutable messages stored in message queues. This variant is suitable for the deterministic (sequential) execution of components with serialized (one by one) event processing, e.g., to avoid reacting to multiple events at the same time (in the same execution cycle; see Section 2.5.2). This mode can be suitable for modeling the integration of independent components whose execution and communication are driven by a central controller, e.g., control token in a local network.

2.5.2 Synchronous systems

The synchronous domain has a notion of logical time and follows the semantics of synchronous programming languages [BB91; Hal+91; EL03]. 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, 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 mode 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 single cycle is not possible, receiver components can process transmitted signals only in the next cycle (initiated by the subsequent tick). For example, logical functions of a controller 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.

Cascade The cascade model supports the execution of components in a *linear way* (one after another in a specific order during a cycle) in contrast 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, enabling communication between components during a cycle in a feed-forward way. In addition, *repeated execution* of components during a single cycle is supported. For example, a single logical controller may implement filters on its inputs and outputs, inducing an execution order. Pipeline-like software or hardware implementations are best modeled with this variant.

Chapter 3

Mixed-semantic composition and verification of reactive components

Statecharts [Har87] are an expressive and widely used modeling formalism to model the dynamic behavior of reactive components, i.e., components that process external stimuli and react to them according to their internal states. However, there exist several statechart language variants that support slightly different operational semantics for the same model elements, e.g., the execution of orthogonal regions. Accordingly, different tools and environments may interpret the same statechart models differently (considering different operational semantics), hindering information exchange and opening the way for inconsistencies in the system design.

In order to mitigate the design of complex reactive systems, different platform- or component-based design techniques [Nuz+15] can be applied by means of the hierarchical integration of standalone components. Nevertheless, existent modeling standards poorly support the integration (composition) of components while considering semantic constraints, especially if *different* execution and interaction semantics have to be employed at different integration levels of the system. For instance, a hardware-related controller may be decomposed into several subcomponents executed concurrently that communicate synchronously via signals (e.g., on a microcontroller's bus), whereas the emergent controller component is integrated into a distributed system where components run in parallel and interact with queued messages. Thus, the design and verification of the composite system necessitate precise modeling languages both at component level and integration level.

The goal of this chapter is to provide a common foundation for the component-based design and formal verification of reactive systems in the form of a configurable and extensible modeling framework. Accordingly, the chapter presents the Gamma Statechart Composition Framework. The framework provides a configurable *statechart* language with semantic variation points to capture various dynamic behaviors (statechart variants) based on the same set of model elements. It also supports mixed-semantic hierarchical integration of heterogeneous statechart components based on different execution and interaction modes (synchronous and asynchronous semantics) based on a *composition* language. The automated formal verification of the created functional design models are realized by semantic-preserving *model transformations* into the input languages of hidden model checker tools and the verification results are automatically back-annotated to the source models.

The rest of the chapter is structured as follows. Section 3.1 gives an overview of the Gamma Statechart Composition Framework, which serves as a common foundation for the results presented in this dissertation. Section 3.2 presents the framework's statechart language, focusing on the supported semantic variation points of statecharts. Section 3.3 formally presents the composition modes of the

framework's composition language. Section 3.4 introduces the model transformations that map the high-level composition and statechart models into low-level analysis models to support their formal verification. Section 3.5 presents the component integration and formal verification workflow in the Gamma framework, which incorporates the modeling languages and verification functionalities presented in the previous sections and thus, enables model checking of integrated state-based models as a reusable functionality. Section 3.6 presents a case study with the framework. Section 3.7 presents related work. Finally, Section 3.8 closes the chapter with concluding remarks and outlines potential directions for future work.

3.1 Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework is our open source integrated tool suite that supports model-based design, validation, formal verification and code generation for component-based reactive systems. It has been developed at the Critical Systems Research Group since 2016. Its goal is to provide a common but extensible foundation for the component-based development of composite reactive systems. Accordingly, its elements and extensions are defined by the contributions presented in this, and subsequent chapters (Chapters 4 and 5). As a general overview of the framework, Figure 3.1 depicts its model transformation chains, 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.

Statechart language The Gamma Statechart Language (GSL – also referred to as the framework's statechart language) is a UML/SysML-based modeling language that supports the definition of (from a composition point of view) atomic components in the form of statecharts. The language plays a central role in supporting the design of components in a flexible way as it enables the import of component models (so-called engineering models) from integrated modeling tools (modeling front-ends), e.g., Yakindu, MagicDraw or SCXML; thus, it can also be regarded as a common formalism into which external models can be transformed. GSL features a textual syntax, as well as a powerful action language and *semantic variation points* that specify and enable the configuration of different semantics for model elements (see Section 3.2).

Composition language Composite components can be created by composing atomic components and other composite components based on well-defined interfaces using the textual Gamma Composition Language (GCL – also referred to as the framework's composition language). *Components* communicate with *events* via well-defined *ports*, which can be connected in a composite component using *channels*. In addition, *port bindings* can be used to map the ports of a composite component to the ports of contained components (component instances). As a special feature, the language supports different *composition modes* (see Section 2.5 for their informal introduction and Section 3.3 for formalization) to control the execution and interaction of contained components.

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

Model validation *Validation of models* takes place at the level of atomic components as well as at integration level by evaluating well-formedness constraints. Altogether, around 240 well-formedness constraints are supported that describe, e.g., nondeterminism between transitions in the statecharts, the checking of model imports, port bindings, values bound to parameters of components, channel constructions, as well as event references in message queues in asynchronous models.

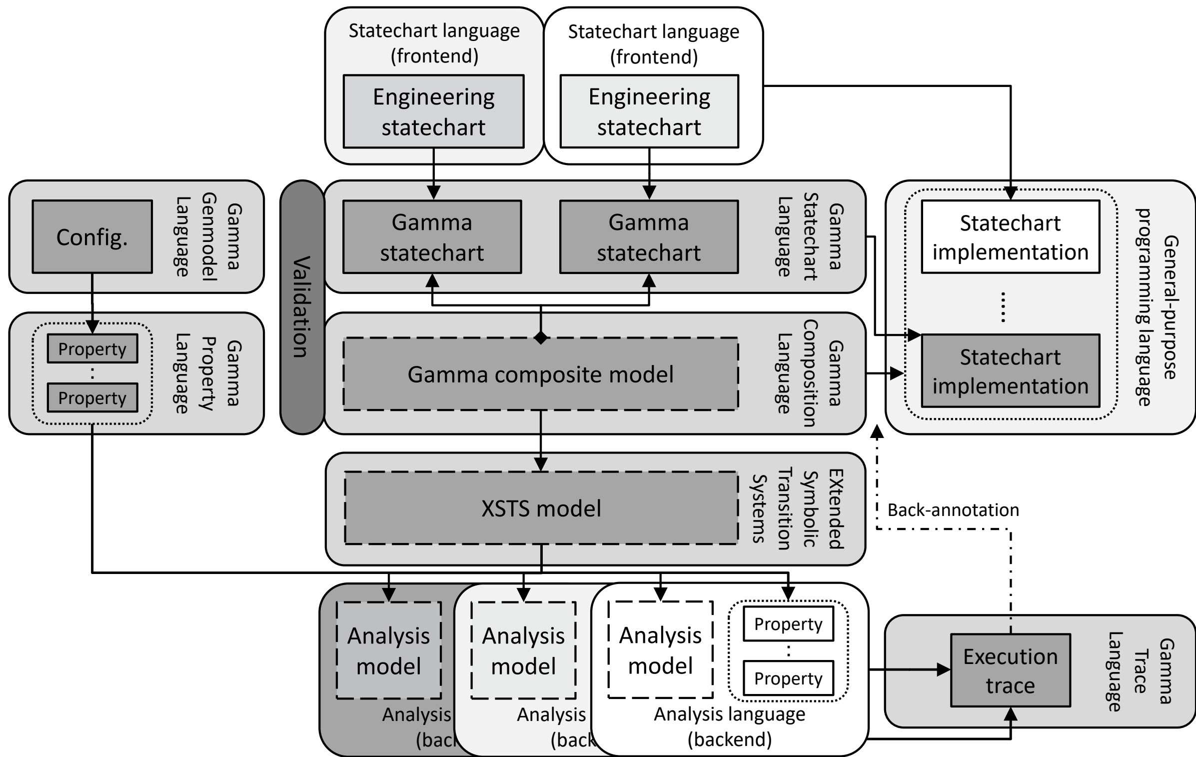

Figure 3.1: Model transformation chains and languages of the framework. Rectangles represent models: solid border represents an atomic model, whereas dashed border represents a composite model. Dotted rectangles represent a set of models belonging together to fulfill a more general purpose. Rectangles with moderately rounded corners represent languages. Rectangles with extensively rounded corners represent functionalities 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 data transformation between the source and target models.

Code generation Once the entire system is modeled as a composition of statechart-based components, the framework can *generate implementation* for standalone statecharts, as well as *composition code* that wraps and connects (based on port bindings and channels) the existing (autogenerated) source code of atomic components based on well-defined interfaces. Accordingly, external code generators (e.g., Yakindu) for statechart components can be integrated by implementing a plugin for the code generator that wraps the external code behind the interfaces generated by Gamma. Currently, the framework supports the generation of Java code [Gra18].

In order to support the step-by-step simulation of component implementations in a type- and interface-independent way, the framework supports a *reflective* API via which the encapsulated objects and functionalities of the component implementations can be accessed using string parameters. The functions of the API are summarized in Table 3.1 The API supports the input of scheduling calls and input events from the environment in addition to retrieving raised output events, variable values and state configurations of the underlying implementations to show internal state. For the flexible and precise simulation of time, a *virtual timer* implementation supporting multiple interfaces is generated.

Table 3.1: Functions of the reflective API for component implementations and their support in terms of component type; atomic statechart components (AS), composite components (CC) or both (BC).

API function (ReflectiveInterface)	Support
String[] getPortNames()	BC
String[] getInputEventNames(String portName)	BC
String[] getOutputEventNames(String portName)	BC
String[] getParameterNames(String portName, String eventName)	BC
String[] getContainedComponentNames()	CC
String[] getRegionNames()	AS
String[] getStateNames(String regionName)	AS
String[] getVariableNames()	AS
boolean isEventRaised(String portName, String eventName)	BC
Object getParameterValue(String portName, String eventName, String parameterName)	BC
ReflectiveInterface getContainedComponent()	CC
Object getVariableValue(String variableName)	AS
boolean isStateActive(String regionName, String stateName)	AS
void reset()	BC
void schedule()	BC
void raiseEvent(String portName, String eventName, Object[] arguments)	BC

Formal verification *Formal verification* of *composite models* is provided by model checking [Cla+18], a technique that explores the behavior of the given model exhaustively with respect to properties specified in mathematical logic (e.g., to check that unsafe states are not reachable). In order to assist engineers, the framework hides the inherent complexity of formal verification by (1) offering a pattern-based approach to specify the required properties [DAC99], (2) using automated semantic-preserving model transformations via the XSTS formalism to the analysis models and queries of verification back-end tools, e.g., UPPAAL, Theta and Spin, and (3) back-annotating [Heg+10] the execution traces retrieved by the model checker (i.e., witnesses or counterexamples for the checked properties) to the composite model to aid their interpretation.

3.2 GSL and its semantic variation points

The Gamma Statechart Language (GSL) serves as a common representation language for component statecharts and supports different *semantic variation points* (statechart semantics) by means of annotations. As abstract syntax of GSL, the model elements and their possible interrelations (which are based on the elements of UML/SysML statecharts) are given by the metamodel in Figure 2.1. The concrete syntax of the language is presented in Tables 3.2, 3.3, 3.4, 3.6 and 3.5 as grammar rules, and illustrated in Figure 3.2, which describes the *spacecraft* component of Figure 2.4 in GSL.

GSL, like every modeling language in Gamma, organizes models into packages and supports their import using a relative path, e.g., interfaces, serving as realizable communicational contracts in the form of ports (see details in Section 3.3), constant declarations and user-defined types (enumerations and records). The language features variable declarations of type boolean, integer, double and custom type (enumeration) which can be organized into records and arrays, as well as timeout declarations. Regarding the description of hierarchy-based behavior, the language supports (orthogonal) regions with (potentially hierarchical) states, as well as fork, join, choice and merge states (pseudo sates) to support defining composite transitions.

Table 3.2: Textual concrete syntax of GSL statecharts.

<table>
<tr><th>Model element</th><th>Syntax rule</th></tr>
<tr><td>Package</td><td>’package’ ID
Import
Statechart*</td></tr>
<tr><td>Import</td><td>’import’ STRING</td></tr>
<tr><td>Statechart</td><td>StatechartAnnotation* // Sets semantic variation points (SVP-*)
’statechart’ ID ’(’ ParameterDeclaration* ’)’ ’[’ Port* ’]’ ’{’
VariableDeclaration*
TimeoutDeclaration*
Region*
Transition*
’}’</td></tr>
<tr><td>ParameterDeclaration</td><td>ID ’:’ Type</td></tr>
<tr><td>Port</td><td>’port’ (’requires’ | ’provides’) ID // Interface reference</td></tr>
<tr><td>VariableDeclaration</td><td>VariableDeclarationAnnotation* ’var’ ID ’:’ Type (’=’ Expression)?</td></tr>
<tr><td>VariableDeclarationAnnotation</td><td>’@Resettable’ | ’@Transient’</td></tr>
<tr><td>TimeoutDeclaration</td><td>’timeout’ ID (’,’ ID)*</td></tr>
<tr><td>Region</td><td>’region’ ID ’{’
StateNode*
’}’</td></tr>
<tr><td>StateNode</td><td>State | PseudoState</td></tr>
<tr><td>State</td><td>StateAnnotation // Supports language extensions: see Section 4.3
’state’ ID ’{’
(’entry’ ’/’ Action+)?
(’exit’ ’/’ Action+)?
Region*
’}’</td></tr>
<tr><td>PseudoState</td><td>’initial’ ID | ’shallow history ID | ’deep history’ ID | ’choice’ ID |
’merge’ ID | ’fork’ ID | ’join’ ID</td></tr>
<tr><td>Transition</td><td>(’@(’ ID ’)’ /* Transition id */)? ’transition’ (’(’ Expression // Priority ’)’)?
’from’ ID ’to’ ID /* StateNodes */ ’when’ Trigger (’[’ Expression ’]’)? (’/’ Action*)?</td></tr>
<tr><td>Type</td><td>SimpleType | ArrayType</td></tr>
<tr><td>SimpleType</td><td>’boolean’ | ’integer’ | ’double’ | ID // Enumeration or record reference</td></tr>
<tr><td>ArrayType</td><td>Type ’[’ Expression ’]’ // Capacity</td></tr>
</table>

In order to facilitate the flexible handling of event parameters, the language supports *transient* (default) and *persistent* events (see Table 3.3). In the former case, the parameter values of a received event are stored only for a *single execution cycle* (step); after that, the values are reset and cannot be retrieved in the following cycles. In the latter case, the parameter values are stored *indefinitely* until the parameter of a new received event overwrites them.

In order to support verification, the language features *resettable* and *transient* variables by means of annotations. *Resettable* variables are reset to the default value of their type at the *beginning* of the execution of the statechart component, which is useful to limit the validity of a variable value to a single execution cycle (step). *Transient* variables are reset to the default value of their type at the *end* of the execution of the statechart component (before entering a permanent system state), which is useful in the case of (temporary) auxiliary variables holding no information in permanent states. These options can be set using the *@Resettable* and *@Transient* variable annotations either manually (by the user) in the GSL model or by specific (pre)processing steps (see Section 3.5.2). Note that, contrary to event parameters where the transient and persistent settings affect the handling and availability of

Table 3.3: Textual concrete syntax of custom type, record, interface and event declarations.

Model element	Syntax rule
CustomTypeDeclaration	'type' ID ':' 'enum' '{' ID* /* Literals */ '}'
RecordTypeDeclaration	'record' ID ':' 'record' '{' (ID ':' Type)* /* Fields */ '}'
InterfaceDeclaration	'interface' ID ('extends' Interface*)? '{' EventDeclaration* '}'
EventDeclaration	('transient' | 'persistent')? ('in' | 'out' | 'inout' | 'internal') 'event' ID ('(' ParameterDeclaration* ')')?

parameter values via the component's public interface (ports) and thus, inter-component communication, these options are set using annotations, highlighting them as an additional, useful feature in the language targeting internal variables to reduce the state space of components and facilitate formal verification.

GSL features a powerful action language that can be utilized to handle variables in the effects of transitions, as well as entry and exit events of states (see Table 3.4). In addition to the *raising* (sending) of *events* and *assignment* statements, the language supports the *declaration* of *local variables*, as well as the sequential execution of statements organized into *blocks*. Deterministic branching can be expressed by *if-else* and *switch* statements, while the language also supports *nondeterministic choices*. *For loops* can be used to express iterative behavior (e.g., array handling) while reusable code can be organized into *functions* and called from multiple places.

As a special feature, the language offers various annotations to select *semantic variation points* to define the semantics of certain modeling elements (see Table 3.5). The informal overview of the semantic variation points (corresponding to the SVP-* notations in Section 2.2) are as follows (see Section 3.4 for the detailed handling of SVP):

- **SVP-1**: the evaluation of enabled transitions (guard expressions) in orthogonal regions, which can take place *before* executing any action in the orthogonal regions (default case in the case of UML/SysML semantics as presented in Section 2.2) or *on the fly*, i.e., the evaluation occurs one by one for each orthogonal region before executing any action in the orthogonal region

Table 3.4: Textual concrete syntax of GSL actions.

Model element	Syntax rule
Action	Block | Statement
Block	'{' Action* '}'
Statement	VariableDeclarationStatement | IfStatement | SwitchStatement | ChoiceStatement | ForStatement | FunctionCallStatement | AssignmentStatement | RaiseEventStatement
VariableDeclarationStatement	'var' ID ':' Type ('=' Expression)?
IfStatement	'if' '(' Expression ')' Action ('else' Action)?
SwitchStatement	'switch' '{' ('case' Expression ':' Action)* '}'
ChoiceStatement	'choice' '{' ('branch' Expression ':' Action)* '}'
ForStatement	'for' '(' ParameterDeclaration 'in' Expression ')' Action
FunctionCallStatement	ID /* Function declaration reference */ '(' Expression* ')'
AssignmentAction	ID /* Variable declaration reference */ ':=' Expression
RaiseEventAction	'raise' ID /* Port */ '.' ID /* Event */ ('(' Expression*')')? // Arguments

Table 3.5: Textual concrete syntax of GSL annotations (semantic variation points).

Model element	Syntax rule
StatechartAnnotation	GuardEvaluation | RegionSchedule | TransitionPriority | OrthogonalRegionSchedule
GuardEvaluation	'@GuardEvaluation' = 'beginning-of-cycle' | 'on-the-fly'
RegionSchedule	'@RegionSchedule' = 'bottom-up' | 'top-down
TransitionPriority	'@TransitionPriority' = 'off' | 'order-based' | 'value-based'
OrthogonalRegionSchedule	'@OrthogonalRegionSchedule' = 'sequential' | 'unordered' | 'parallel'

but (potentially) after the execution of other actions in other orthogonal actions in accordance with the order specified in SVP-4;

- **SVP-2**: conflict resolution between transitions in the case of parent and child regions of hierarchical states, which can be *bottom-up* (default) or *top-down* resolution;
- **SVP-3**: *priority* between enabled transitions leaving the same state: the *absence* of priority leads to nondeterministic choices between enabled transitions during execution (default), whereas the *order-based* and *value-based* settings define priorities between them in accordance with their definition order (an "earlier" definition means a higher priority) or their assigned priority value (a higher value means a higher priority), respectively;
- **SVP-4**: the execution of actions in orthogonal regions of a composite state, which can be *sequential*, i.e., in the order of the declaration of regions, *unordered*, i.e, any region permutation (without interleavings) is considered valid, and *parallel*, i.e., actions in orthogonal regions can interleave in any way (default).

In order to allow for the description of synchronous systems (e.g., hardware-related controllers or hardware descriptions based on sequential networks), the statechart language supports *signal-based* event handling (synchronous execution) introduced in [Wag92] by means of complex triggers, i.e., the logical combination (Not, as well as And, Or and Xor) of sampled signals represented by input events (see Table 3.6). Accordingly, transitions can be directly triggered by the *ticks* of the global execution clock (*on-cycle* trigger) or timeouts in addition to events received via ports. As a syntactic sugar, the language also supports referencing all event sources using a single trigger (*any* trigger). Nevertheless, in addition to supporting synchronous semantics, statechart components can be adapted by means of composition techniques (see *asynchronous adapters* in Section 3.3.7) to conform to the message-based (asynchronous) execution semantics present, for example, in UML/SysML statecharts.

The language is given a formal (denotational) semantics by means of model transformations that

Table 3.6: Textual concrete syntax of GSL triggers.

Model element	Syntax rule
Trigger	BinaryTigger | NotTrigger | ParenthesesTrigger | SimpleTrigger | OnCycleTrigger
BinaryTigger	Trigger ('&&' | '||' | '^') Trigger
NotTrigger	'!' Trigger
ParenthesesTrigger	'(' Trigger ')'
SimpleTrigger	PortEventTrigger | TimeoutTrigger | AnyTrigger
PortEventTrigger	ID /* Port */ '.' ID // Event declaration
TimeoutTrigger	'timeout' ID // Timeout declaration
AnyTrigger	'any'
OnCycleTrigger	'cycle'

```
package spacecraft
import "Interface/Interface.gcd" // Importing interfaces to reference them in port declarations
// Annotations for setting semantic variation points (SVP-*)
...
statechart Spacecraft [
 port Connection : provides DataSources // Ports for communication
] {
 // Variable and timeout declarations
 var battery : integer := 100
 var data : integer[8]
 timeout recharge, consume, transmit
 // Regions and contained state nodes (potentially with entry and exit actions in states)
 region Main {
   entry MainEntry
   state Spacecraft {
   region Communication {
    initial CommunicationEntry
    state WaitingPing {
     entry / for (i : integer in 0 .. 7) { data[i] = 0; }
    }
    state Transmitting
   }
   region BatteryManagement {
    initial BatteryManagementEntry
    state NotRecharging
    state Recharging
   }
  }
 }
 // Transitions with source and target nodes, (a combination of) triggers, guards and effects
 transition from MainEntry to Spacecraft
 transition from CommunicationEntry to WaitingPing
 transition from WaitingPing to Transmitting when Connection.ping [Connection.ping::types == ::TRANSMIT]
 transition from Transmitting to WaitingPing when timeout consume [battery < 40]
 transition from Transmitting to WaitingPing when timeout transmit [allTransmitted(data) or battery < 40]
 transition from BatteryManagementEntry to NotRecharging
 transition from NotRecharging to Recharging when timeout consume [battery < 80]
 transition from Recharging to Recharging when timeout recharge /
  if (battery < 100) { battery := battery + 1; }
 transition from Recharging to NotRecharging when timeout recharge [battery = 100]
}
```

Figure 3.2: GSL representation of the *spacecraft* component presented in Figure 2.4.

map GSL models into the XSTS formalism (see Section 3.4). This way, the *transition function* belonging to GSL components (i.e., atomic GCL components) is given in terms of XSTS model elements (see *trans* element in Figure 2.7), i.e., the execution of components is mapped into the level of state machines (see Section 2.2) by these model transformations.

As a demonstration of the language's flexibility, we *integrated* Yakindu in the framework as a modeling front-end for state-based components via a semantic-preserving automated *model transformation* that maps Yakindu statecharts into the statechart language [Gra16][*e*12]. Namely, the statechart language of Yakindu also builds on UML/SysML but has different execution semantics as it relies on *top-down* conflict resolution between parent and child regions, the *sequential* execution of actions in orthogonal regions, distinct *priorities* between transitions leaving the same state and *on the fly* evaluation of enabled transitions in orthogonal regions; each of which can be handled without any limitation by our framework's statechart language.

3.3 GCL and the formal semantics of its composition modes

This section presents the formal structures and semantics of the Gamma Composition Language (GCL), distinguishing it from other informal and semi-formal modeling languages. In addition to the overview of the supported model elements, the subsections include short discussions about additional practical and theoretical aspects, design decisions and consequences on formal verification and code generation.

The section starts with an overview of the syntax of GCL (Section 3.3.1), focusing on asynchronous adapters and composite components. Section 3.3.2 formally defines events and related structures, then the syntactic definition of synchronous components (Section 3.3.3) is introduced. Next, synchronous composite components and cascade composite components are formalized both syntactically and semantically (Sections 3.3.4 and 3.3.5). After defining asynchronous components (Section 3.3.6), asynchronous adapters and their semantics are presented (Section 3.3.7). The section proceeds with the definition of asynchronous composite components (Section 3.3.8) and their semantics in terms of their environment, as well as received and sent messages, occurrences and execution traces. Finally, the section concludes with the definition of scheduled asynchronous composite components (Section 3.3.9). In order to help the reader find their way through the following pages, Appendix B lists the symbols used in the definitions along with a short description.

3.3.1 GCL syntax

GCL features *asynchronous adapters* and various *composite components* to build synchronous and asynchronous systems based on the (hierarchical) composition of atomic statechart components defined in GSL (see Section 3.2). As a special feature, GCL also supports *message demultiplexing* in asynchronous adapters to provide additional configurability options for communication during component composition.

Asynchronous adapters Figure 3.3 depicts an excerpt of the model elements regarding *asynchronous adapters* and their possible interconnections supported by GCL (metamodel excerpt disregarding *triggers*, which are detailed in Table 3.6); the related textual syntax is presented in Table 3.7.

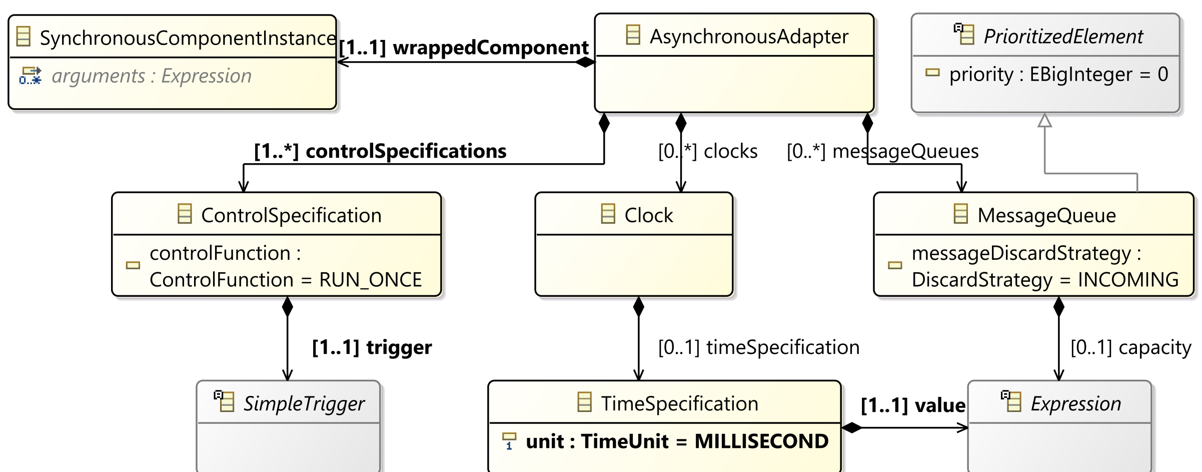

Figure 3.3: Elements of *asynchronous adapters* in GCL (metamodel excerpt).

```
adapter AynchronousSpacecraft of component spacecraft : Spacecraft [
 // Additional control port on the adapter
 port control : Control
] {
 // Clock emitting a tick every millisecond
 clock millisecClock(rate = 1 ms)
 // Run wrapped component upon any kind of received event
 when any / run
 // Connection messages are placed in a higher priority queue
 queue protocolMessages (priority = 2, capacity = 4, discard = incoming) {
  connection.any
 }
 // Other control messages are placed in a lower priority queue
 queue controlMessages (priority = 1, capacity = 2, discard = oldest) {
  control.any, millisecClock
 }
}
```

Figure 3.4: Asynchronous adapter wrapping the synchronous *spacecraft* statechart component.

Each synchronous adapter (see an example in Figure 3.4) wraps a single synchronous component instance, turning it into an asynchronous component. An adapter implicitly has all *ports* of the wrapped component, and can optionally define additional ones to receive control messages (see below).

As a primary feature, asynchronous adapters can have one or more *message queues*, which store incoming messages 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, then the situation is handled according to the *message discard strategy* attribute.
- *Message discard strategy* can be either *incoming* or *oldest*, specifying that in case a message is sent to an already *full* queue, the incoming message, or the one that has been in the queue for the longest period (at index 0) shall be discarded.
- *Priority* of a queue specifies the order in which the contents of message queues are retrieved during the execution of the asynchronous component (a greater value means higher priority).

Table 3.7: Textual concrete syntax of GCL asynchronous adapters.

Model element	**Syntax rule**
AsynchronousAdapter	'adapter' ID 'of' 'component ID : ID /* Adapter synchronous component */ '(' ParameterDeclaration* ')' '[' Port* ']' '{' ClockDeclaration* ControlSpecification* MessageQueue* '}'
ClockDeclaration	'clock' ID '(' 'rate' '=' Expression ('s' | 'ms')')'
ControlSpecification	'when' Trigger '/' ('run' | 'reset')')'
MessageQueue	'queue' ID '(' 'priority' '=' Expression ',' 'capacity' '=' Expression (',' 'discard' '=' ('incoming' | 'oldest'))? ')' '{' (ID /* Clock reference */ | ID.ID /* Port event reference */ | ID.'any' /* Any port event reference */)+ '}'

- *Event references* specify the types of messages that can be stored in the particular message queue (demultiplexing incoming messages into message queues).

During execution, messages are retrieved from the message queues *one by one* (message processing). 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 trigger different execution of the wrapped component (recall that *any triggers* are a shortcut to refer to every incoming message): if a specified message arrives, the wrapped component is *executed* or *reset* to its initial state according to the specified setting. 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 specification is specified for the corresponding message.

Finally, asynchronous adapters can contain *clocks*, which emit tick events at defined timed intervals based on a *rate* attribute. Tick events also have to be assigned to a queue and can be handled in control specifications similarly to regular events received from ports.

As an example, Figure 3.4 describes an asynchronous adapter wrapping the synchronous *spacecraft* statechart component (presented in Figure 2.4) by introducing another control port, a clock with a rate of 1 ms, an *any* control specification and two message queues with different attributes.

Composite components Figure 3.5 depicts an excerpt of the model elements regarding *composite components* and their possible interconnections supported by GCL (metamodel excerpt disregarding concrete composite component types); the related textual syntax is presented in Table 3.8.

Every *composite component* (synchronous, cascade, asynchronous and scheduled asynchronous) consists of *component instances* (also referred to as constituent components), which communicate via well-defined *ports* (identical to statechart ports) and thus, define the emergent behavior of the composite component. Ports can realize interfaces either in *provided* or *required* mode (see the *Port* entry in Table 3.2): in the former case, the transmission of events is in accordance with the declared direction (see the *EventDeclaration* entry in Table 3.3), whereas in the latter case, the directions are "inverted." Ports that realize the same interface in different modes (provided and required) can be connected

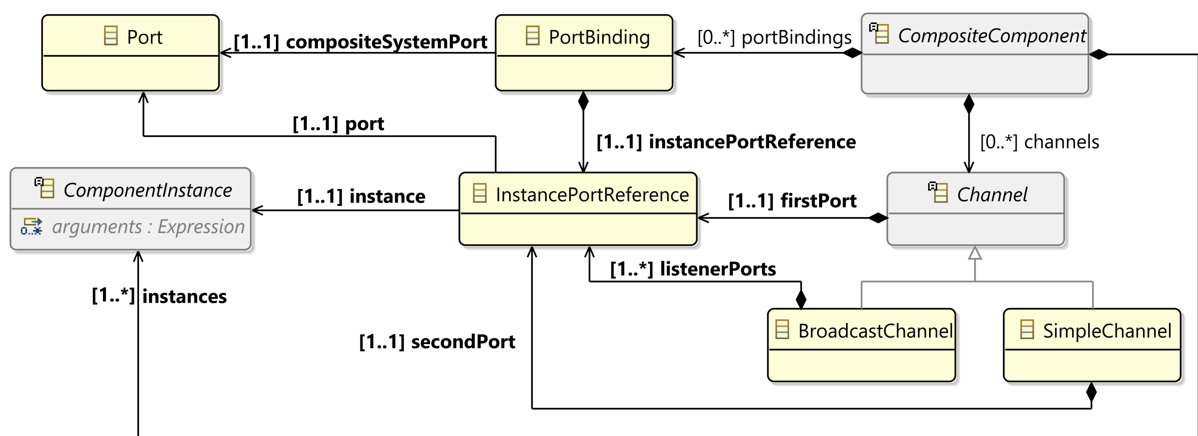

Figure 3.5: Elements of *composite components* in GCL (metamodel excerpt).

```
[sync / cascade / async / scheduled-async] SimpleSpaceMission [
 port control : requires StationControl
] {
 // Declaring the station and spacecraft component instances
 component station : GroundStation
 component spacecraft : Spacecraft
 // Exporting the station's control port (port binding)
 bind control -> station.control
 // Connecting the ports of the spacecraft and the station
 channel [spacecraft.connection] -o)- [station.connection]
}
```

Figure 3.6: GCL model of the *simple space mission* with a *ground station* and a *spacecraft* component.

by *simple channels* (one-to-one port connections between component instances using instance port references) or *broadcast channels*, which support the distribution of events from a single source port to multiple connected ones. In order to "export" the ports of a component instance to the environment (i.e., the boundary of the composite component), the language features *port bindings*, this way, allowing for the hierarchical composition of the components.

Figure 3.6 describes the textual GCL model of the *simple space mission* (presented in Section 2.3) including every supported composition mode. As illustrated, the definition of composite models in different composition modes differ only in a single keyword (*sync*, *cascade*, *async* or *scheduled-async*); the definition of *ports*, *component instances*, *port bindings* and internal *channels* are identical in each composition mode. Therefore, users can tailor the integration and verification process according to their needs and expectations about their system's behavior in a flexible way.

Table 3.9 summarizes the component types supported by the GCL in terms of *synchrony* and *compositeness*.

Table 3.8: Textual concrete syntax of GCL composite components.

Model element	Syntax rule
AbstractCompositeComponent	SynchronousCompositeComponent | CascadeCompositeComponent | ScheduledAsynchronousCompositeComponent | AsynchronousCompositeComponent |
SynchronousCompositeComponent	'sync' CompositeComponent
CascadeCompositeComponent	'cascade' CompositeComponent
ScheduledAsynchronousCompositeComponent	'scheduled-async' CompositeComponent
AsynchronousCompositeComponent	'async' CompositeComponent
CompositeComponent	ID '(' ParameterDeclaration* ')' '[' Port* ']' '{' ComponentInstance* PortBinding* Channel* ExecutionList? // For cascade and scheduled-async '}'
ComponentInstance	'component' ID ':' ID '(' Expression* ')' // Component type reference
PortBinding	'bind' ID /* Composite component port */ '->' ID '.' ID // Component instance's port
Channel	'[' ID.ID ']' '-o)-' '[' ID.ID (ID.ID)* ']' // Component instances' ports
ExecutionList	(ID.ID)* // Sequence of component instances

Table 3.9: Component types supported by GCL.

	Atomic	Composite
Synchronous	Statechart	Synchronous composite component Cascade composite component
Asynchronous	Asynchronous adapter	Scheduled asynchronous composite component Asynchronous composite component

Message demultiplexing in asynchronous adapters and composite components In order to support *message queues* shared among multiple components, GCL features *message demultiplexing* in asynchronous adapters to allow for defining what signals a message is converted into during message processing. The *risGlobalQueue* component in Figure 3.7 shows an excerpt of the asynchronous adapter that wraps the components of the RIS model (presented in Section 2.3) and demultiplexes messages in the *globalQueue*, e.g., from the *Out* port of the *control center* to the *In* port of the *dispatcher*. As depicted, demultiplexing is defined using the -> operator in the body of a queue definition: its left hand side specifies a message type (asynchronous event received via an adapter port) the message queue stores, whereas the right hand side defines a set of signals (synchronous events via the wrapped component's port) that the message is converted into during its processing; multiple signals can be specified using & symbols. Note that the events of the left and right hand side must have parameter declarations with the same type. As syntactic sugar, GCL supports the *any* keyword to reference each event of a port. Also, the -> operator and its right hand side can be omitted if the two sides are equal.

```
adapter risGlobalQueue of component ris : RIS { // Wrapped component
 // Single control specification: execute once upon any incoming message
 when any / run
 // Shared global queue
 queue globalQueue(priority = 2, capacity = 6, discard = incoming) {
  controlCenterOut.rigel -> dispatcherIn.rigel, // Demultiplexing a single message
  dispatcherOut.any -> objectHandler.any, // Demultiplexing all messages of a port
  ...
 }
}
// Cascade composite component (CCC)
cascade RIS [
 // Internal ports of the CCC
 port controlCenterIn : requires Rigel
 port controlCenterOut : provides Rigel
 ... // Additional in and out ports of other components
] {
 // Statechart components of the RIS model
 component controlCenter : ControlCenter
 component dispatcher : Dispatcher
 component objectHandler : ObjectHandler
 // Binding component ports to the ports of the CCC
 bind controlCenterIn -> controlCenter.In
 bind controlCenterOut -> controlCenter.Out
 ...
}
```

Figure 3.7: RIS model (depicted in Figure 2.6) variant excerpt with a *shared global queue* that utilizes *message demultiplexing*.

Message demultiplexing provides configurability options for component integration and allows for capturing structure-related aspects (port connections). For example, the feature can be utilized to define a *shared* global message queue for a composite component that stores messages sent among contained components (as depicted in Figure 3.7). Such a solution can be modeled featuring message demultiplexing in GCL as follows:

1. a single cascade composite component (CCC) is defined that contains every necessary component while also specifying their execution order (channels between the components are not defined);
2. the CCC declares *internal* ports for each component, i.e., ports that transform raised events into received events for inter-component communication;
3. the CCC is wrapped by an asynchronous adapter that defines a single message queue with the required attributes and message demultiplexing for the internal ports: the left hand side will contain the sender port, whereas the receivers will be on the right hand side.

3.3.2 Events, event vectors and event sequences

This section formally defines *events*, as well as *event vectors* used by synchronous components, and *event sequences* used by asynchronous components.

3.3.2.1 Events

The following definitions 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 3.1.** An *event* is an observable phenomenon that can occur, e.g., message reception or the change of situation (state). Given a set of events E, the finite domains of event parameters are defined by the domain function $\mathcal{D} : E \rightarrow 2^{\{d_1,\dots,d_n\}}$. 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, $inst_\bot(e)$ is defined as $inst(e) \cup \{(e, \bot)\}$, where $(e, \bot)$ 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 $inst_\bot(E)$ similarly). ▪

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

3.3.2.2 Event vectors

Synchronous 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 3.2.** 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 $v_E(e) \in inst_\bot(e)$. The set of all possible event vectors is denoted by V_E. ▪

Discussion Event vectors need memory to be represented 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 their parameter domains' size thus directly affect the state vector's size in formal verification or the memory requirements of an implementation.

3.3.2.3 Event sequences

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

> **Definition 3.3.** 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 $S_\sigma(A)$. ▪

3.3.3 Synchronous component

The following definition specifies the formal syntactic contract of synchronous components. A synchronous component shall 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 3.4.** A synchronous component is a tuple $\ominus = (S, s^0, I, O, \mathcal{D}, T)$:
> - S is the set of potential states, with $s^0 \in S$ being the initial state.
> - I is the set of input events and O is the set of output events such that $I \cap O = \emptyset$. The set of all events is denoted by $E = I \cup O$.
> - $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$ is the domain function of the events.
> - $T \colon S \times V_I \to S \times V_O$ is the transition function determining the component's next state and the output event vector when executing it in a given state with an input event vector. Note that this definition requires the component to have a *deterministic* behavior.[a] ▪
>
> [a] Naturally, the definitions could be extended to nondeterministic models.

If $\ominus$ is a timed component, we allow S to track the values of *clock variables* [BY04], 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* is 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).

Semantics As a main feature of their semantics, 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 [Wag92]. We use this formalism as 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). Also, as events can occur *at the same time*, interleavings introduced by the often arbitrary order of sequential message passing do not have to be analyzed. Nevertheless, the more widespread Event-Driven Finite State Machine, which is the basis of most commonly used statechart formalisms, e.g., UML/SysML, 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 it must be ensured that they are executed *every time* a signal arrives (see the asynchronous adapter in Section 3.3.7).

Discussion As a main feature, this definition describes an abstract behavioral contract that can be implemented by multiple formalisms, and this way, the framework can support the integration of external tools (modeling front-ends) and formalisms. Accordingly, an atomic statechart defined in GSL is considered as an atomic synchronous component. In addition, the XSTS formalism presented in Section 2.4 allows for formally defining different semantic variations for reactive behavior, providing an intermediate-level behavior representation. The rest of the section is about the semantics of scheduling components, which is one of the main focuses of this work.

3.3.4 Synchronous composite component

A synchronous composite component is defined by instantiating a set of constituent components (component instances), 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 3.5.** A synchronous composite component is a tuple $\circledS = (\mathbf{C}, I, O, \rightleftharpoons)$:
> - $\mathbf{C} = \{\ominus_1, \ldots, \ominus_K\}$ 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 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 3.6.** A synchronous composite component $\circledS$ is itself a synchronous component $\circledS\rangle\ominus = (S, s^0, I, O, \mathcal{D}, T)$:
> - $S = S_1 \times \ldots \times S_K \times V_{\hat{O}}$ 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.

- $s^0 = (s_1^0, \ldots, s_K^0, \bot_{\hat{O}})$ is the initial state, where every constituent component is in its initial state and the last output event vector $\bot_{\hat{O}} \in V_{\hat{O}}$ assigns $\bot$ to every output event ($\forall e \in \hat{O} : \bot_{\hat{O}}(e) = \bot$).
- I is the set of exported input events and O is the set of exported output events as defined in Definition 3.5 (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.

3.3.5 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 3.7. 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 $\hat{O} = \bigcup_{k=1}^{K} O_k$.
- $\rightleftharpoons : \hat{I} \setminus I \rightarrow \hat{O}$ 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); thus, 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 3.8. 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 3.7 (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 $X[i]$ $(1 \leq i \leq n, n = |X|)$:
  - 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 $e \in I_k$,

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

    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 to observe an output). Second, it is sometimes desired to "decorate" a component with auxiliary components such as adapters or monitors without introducing a delay in the observable effect of an event. Moreover, it is convenient to think in terms of pipelines, which is best expressed with cascade components.

One drawback of using cascade composite components is that the outputs of constituent components may overwrite each other if a particular component is executed multiple times in a single cycle (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.

3.3.6 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 selected from the potential output sequences nondeterministically.

> **Definition 3.9.** An asynchronous component is a tuple $⊜ = (S, s^0, I, O, \mathcal{D}, T)$:
> - S is the set of potential states, with $s^0 \in S$ being the initial state.
> - I is the set of input events and O is the set of output events such that $I \cap O = \emptyset$. The set of all events is denoted by $E = I \cup O$.
> - $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$ is the domain of the events.
> - $T \subseteq S \times inst(I) \times S \times inst(O)^*$ is the transition relation, which determines the possible next states and the possible sequences of output events of the component ($inst(O)^*$) when executing it in a given state with a given input event. Note that this definition *does not* require deterministic behavior. ▪

Semantics Contrary to synchronous components, the semantics of asynchronous components is closest to Event-Driven Finite State Machines or the variant of statecharts defined in UML/SysML. Although currently not supported by GSL, asynchronous components could be described directly by statecharts. In Gamma, the current means of defining an asynchronous statechart component is to define a synchronous component containing a statechart, and wrap it in an asynchronous adapter.

Discussion Similarly to synchronous components, this definition also describes an abstract behavioral contract that can be implemented by multiple formalisms. 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, can be different depending on the internal order of event raisings. This phenomenon poses challenges to both verification and code generation, and hinders the reproducibility of test cases and counterexamples. Nondeterminism, however, is inherent in asynchronous systems anyway.

3.3.7 Asynchronous adapter

An asynchronous adapter wraps a single synchronous component and converts it into the asynchronous domain. To do this, a *trigger predicate* with a set of trigger specifications must be defined. Asynchronous adapters may also introduce ports in addition to the ports of the wrapped component. 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 3.10.** An asynchronous adapter for a synchronous component is defined as a tuple $⊕ = (⊖, e_c, trig)$:
> - $⊖ = (S_s, s_s^0, I_s, O_s, \mathcal{D}_s, T_s)$ is the wrapped synchronous component.
> - e_c is the *control event.*

- $C = \{c_1^{t_1}, \cdots, c_n^{t_n}\}$ is the set of clocks, where $c_i^{t_i}$ produces e_c periodically after every t_i.
- $trig : I_s \cup \{e_c\} \to \{\top, \bot\}$ 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, $trig(e)$ is $\top$, feed the collected messages and emit messages created from the resulting output event vector.

Definition 3.11. An asynchronous adapter ⊕ for a synchronous component is itself an asynchronous component ⊕⟩⊜ $= (S, s^0, I, O, \mathcal{D}, T)$:

- $S = S_s \times v_I$ is the set of potential states; each state consists of the wrapped synchronous component’s state 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 $trig(e) = \bot$, 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 $trig(e) = \top$, 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. 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 Section 3.3.8.2).

3.3.8 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 3.12. An asynchronous composite component is a tuple ⓐ $= (\mathbf{C}, I, O, \rightleftharpoons)$:

- $\mathbf{C} = \{⊜_1, \ldots, ⊜_K\}$ is the set of asynchronous components constituting the composite component, each component being $⊜_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
- $I \subseteq \hat{I}$ 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. ▪

Semantics 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. Accordingly, the semantics of asynchronous composite components is defined in terms of their communication with their *environment* using *messages*, as well as the valid *execution traces* of their constituent components.

3.3.8.1 Environment of the component

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

Definition 3.13. Given an asynchronous composite component ⓐ $= (\mathbf{C}, I, O, \rightleftharpoons)$, its *environment* is a tuple ⓔ $= (E_I^{env}, E_O^{env})$:

- $E_I^{env} = O$ is the input events of the environment that consume the output events of the asynchronous composite component.
- $E_O^{env} = I$ 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. Note that its behavior can be restricted during verification, e.g., by scenario-based contracts (see Section 5.3.1).

3.3.8.2 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 3.14. Given an asynchronous composite component ⓐ with its environment ⓔ, an asynchronous *message* is a tuple $m = (e_O, p, E_I)$:

- $e_O \in \hat{O} \cup E_O^{env} \cup C$ 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.
- $E_I \subseteq \hat{I} \cup E_I^{env}$ is the set of target input events of the message, possibly targeting the environment.
- If $e_O \in E_O^{env}$ then $E_I \subseteq I$ and if $E_I \subseteq E_I^{env}$ 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 ⊕ then $E_I = \{e_c\}$, where e_c is the control event of ⊕. 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 3.15. 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 $src(m) =$ ⊜$_k \in \mathbf{C}$ when $e_O \in O_k$ or $src(m) =$ ⓔ if $e_O \in E_O^{env}$.

Furthermore, let $t = (s, e_I, s', \omega) \in T_k$ be a transition of a constituent component ⊜$_k$. 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 3.14. ▪

Discussion A message is a runtime object, i.e., it has "object identity." 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 3.16. 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 ⊜$_k$ ($e_I \in I_k$ 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. Rule 3) 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. Rule 4, 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 (e.g., unreliable channels, see Section 3.6) or as part of the behavior of the communication-related components.

Regarding time, Rule 5 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, Rule 6 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 Section 3.3.3 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.

3.3.9 Scheduled asynchronous composite component

The syntactic definition of a scheduled asynchronous composite component differs from asynchronous composite components only in the definition of an additional *execution sequence* of constituent components.

Definition 3.17. A scheduled asynchronous composite component is a tuple Ⓢ $= (\mathbf{C}, X, I, O, \rightleftharpoons)$:

- $\mathbf{C} = \{⊜_1, \ldots, ⊜_K\}$ is the set of asynchronous components constituting the composite component, each component being $⊜_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
- $X \in \mathbf{C}^*$ is a finite *ordered sequence* (with potential repetitions) of asynchronous components called the *execution sequence* specifying the execution order of components.
- $I \subseteq \hat{I}$ 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 $e' \in \rightleftharpoons(e)$, $\mathcal{D}(e) = \mathcal{D}(e')$.

▪

Semantics The scheduled asynchronous composite component is the restricted variant of the asynchronous composite component: the execution sequence restricts the valid set of execution traces as follows.

> **Definition 3.18.** Let $E^t = R*$ denote the valid execution traces of scheduled asynchronous composite components projected to a sequence of transition occurrences, where $R = |t'_1|, \ldots, |t'_n|, n = |X|)$ denotes the ordered sequence of transition occurrences (or their absence if no transition is triggered) conforming to the execution sequence of the contained asynchronous composite components: $t'_k \in T_k \cup \bot$. ▪

Discussion The scheduled asynchronous composite component can be considered as the restriction of asynchronous composite components using the characteristics of cascade composite components as the defined execution sequence limits the possible execution orders of the constituent components. From another perspective, it can also be regarded as the extension of cascade composite components by introducing event sequences and messages in place of the event vectors. Accordingly, this component can be used in cases where the execution order of the asynchronous components is known (can be guaranteed) and one by one message processing is required. As an added value, these features can greatly reduce the potentially large state space of asynchronous components, enabling formal verification even when it would not be feasible with asynchronous composite components.

Note that as the execution sequence limits the execution of constituent components, scheduled asynchronous composite components may be composed of only asynchronous adapters or other scheduled asynchronous composite components (valid constituent components).

3.4 Model transformations for formal verification

The role of model transformations that map integrated state-based (statechart) models into analysis models is twofold. First, they allow for the formal verification of these functional models with respect to formalized properties using model checkers, i.e., they integrate verification back-ends to the framework. In addition, they implicitly give denotational semantics to the modeling languages of the framework by mapping the structures of the the statechart language (GSL, see Section 3.2) and composition language (GCL, see Section 3.3) into low-level analysis constructs with well-defined semantics. Recall that Section 3.3 defined the semantics of GCL on the basis of *abstract behavioral contracts* with respect to synchronous and asynchronous components as composite component semantics build on these definitions. Accordingly, these model transformations adopt the GSL language, considering its semantics with the supported semantic variation points, to the formal behavioral contracts of the framework.

The framework currently features four fundamentally different model transformations that map composite models (including the atomic statechart models) into different analysis languages processed by different model checkers. This multitude of model transformations gives flexibility to the framework as the integrated back-ends can complement each other and support different types of input models, e.g., models with different characteristics, such as timed models, models with many inputs or parallel behavior, as well as different verifiable properties, such as CTL* [EH86], or its subsets, LTL or CTL. The currently supported model transformations with an overview of their characteristics are as follows.

- **DU** (direct-UPPAAL) models are derived by directly mapping composite models [Gra16] into the timed automata formalism of the UPPAAL model checker[1] [Beh+06]. The resulting model forms a network of automata, where atomic components (modeled as separate automata) communicate via shared variables and synchronization constructs (channels) supported by the language. System execution is conducted by auxiliary scheduler automata, controlling the execution of instantiated components.
- The **DX** (direct-XSTS) transformation directly maps composite models into the *EXtended Symbolic Transition System* (XSTS) [Mon20] [*j*5] formalism, supported by the Theta model checker[2] [Tót+17]. XSTS, as presented in Section 2.4, is an analysis language with low-level constructs that serves as an intermediate representation for reactive behavior in the framework, enabling the low-effort integration of different model checker back-ends.
- **XU** (XSTS-UPPAAL) models are derived by mapping composite models into an XSTS model in the same way as in the DX transformation, which is then mapped into the timed automata formalism of UPPAAL. In contrast to DU models, which are based on *control locations* and resemble the structure of statecharts, these models can be considered as *control flow automata* (CFA) [LNN] where locations are used to describe the logical structure of transitions (contained actions) and do not hold explicit information about the state of the system.
- The **XP** (XSTS-Promela) transformation, similarly to XU, builds on and extends the results of the DX transformation, which is then mapped into the Promela (process algebra) language of the Spin model checker[3] [Hol11; Hol05] . Contrary to XU, the structure of the resulting models is analogous to that of the source XSTS models as Promela features low-level modeling structures (resembling the elements of imperative programming languages, e.g., C) with semantics close to that of the elements of XSTS. As a special feature, the XP transformation supports two different approaches to handle *message queues* of asynchronous adapters, namely using the native *asynchronous channel* constructs of Promela (XP-N), or using *arrays* (XP-A) [*j*5].

In the following, the section details the DX model transformation, which has a central role in the framework due to its reusability in the other model transformations and thus, the integration of model checker back-ends. As for the other model transformations, the transformation rules mapping GSL and GCL models into UPPAAL automata can be found in [Gra16], whereas the XP transformation is detailed in [*j*5].

The DX transformation comprises two major parts, namely mapping atomic components (GSL statecharts) and the constructs of the composition (GCL model elements) into XSTS, which are covered in Sections 3.4.1 and 3.4.2, respectively, in the context of the *spacecraft* component excerpt presented in Figures 2.4 and 2.7.

3.4.1 Mapping statecharts into XSTS

Mapping GSL statecharts (whose informal semantics is presented in Section 2.2 with extended semantic variations points introduced in Section 3.2) into XSTS models comprises six major parts, namely, mapping (1) variable and timeout declarations, (2) events contained by ports, (3) control structures (regions and states) and (4) transitions, in addition to (5) defining model initialization and (6) the behavior of its environment.

[1] http://www.uppaal.org/
[2] http://theta.inf.mit.bme.hu/
[3] https://spinroot.com/

```
// Original record types
record CompositeRecord {
 // Record array type
 recordArrayField : SimpleRecord[8]
}
record SimpleRecord {
 integerField : integer,
 booleanField : boolean
}
// Corresponding "unrolled" record
record UnrolledCompositeRecord {
 // Arrays of the contained fields
 integerFields : integer[8],
 booleanFields : boolean[8]
}
```

Figure 3.8: Unrolling of composite record types.

Variable and timeout declarations GSL and XSTS support the same primitive variable types (boolean, integer and double), as well as custom (enum) types (see Line 1 in Figure 2.7) in addition to array structures. In such cases, the mapping of the variables is trivial: for a GSL variable, an XSTS variable is created with the corresponding type (see Line 8 in Figure 2.7). In addition, GSL supports the definition of records (structures of encapsulated data) that have to be “unrolled” (“separation” of the contained fields) potentially in a hierarchical way (record fields), i.e., a (potentially one-element) sequence of variables is created for each contained field. Note that the definition of recursive data structures (e.g., linked lists) is not supported. In addition, the unrolling can also handle fields with a *record array* type (see Figure 3.8): the *array of a record* is handled as the *record* of the *arrays* of the *contained fields* in the emergent hierarchical type structure, which can be handled during unrolling in the aforementioned way.

Timeout declarations are mapped into clock variables (see Line 9) to indicate that they measure time and have to be handled differently during verification.

Events Each event (input and output) of each port is mapped into a boolean variable (see Line 4) that stores whether the corresponding event is present (true) or not (false). The parameter declarations of events are mapped in the same way as variable declarations (see Line 5).

Control structures Each region is mapped into (1) a custom type with an *inactive* literal (representing the inactivity of the corresponding state), and additional literals representing the contained states (see Lines 2-3) and (2) a variable with a **ctrl** annotation and as type, the corresponding custom type (see Lines 6-7). In addition, if the region has *history*, the custom type is extended with a literal (history literal) for each state, representing that the last active state in the region before deactivation was the corresponding state.

Transitions During the mapping of transitions, (1) every transition in the statechart is mapped into a sequence of statements, which are (2) composed according to the control structure hierarchy and the selected semantic variations (see Section 3.2) to model a single execution cycle (step) of the statechart (see Phases 1-4 of the statechart semantics presented in Section 2.2). The emergent structure serves as the *transition function* of the state machine captured in the form of a statechart using GSL.

The (potentially composite) *trigger* of a transition is mapped into (the combination of) references to the corresponding boolean variables (event references), and its *guard* expression (if any) is directly mapped into a boolean expression. The conjunction of the two parts, along with a reference to the source of the transition (i.e., to check whether the corresponding variable is set to the literal corresponding to the source state), is wrapped in an assume statement (see e.g., Lines 25 and 34) in accordance with Phase 1 of the statechart semantics in Section 2.2 (transition *enabledness*).

The *effects* of the transition (Part 2 of transition firing defined in Phase 3) are mapped directly into XSTS statements in accordance with Table 3.10 (see Line 39) with the exception of function calls; as XSTS does not (yet) support functions, function calls are inlined and mapped accordingly.

With the trigger, guard and effects being mapped, the corresponding sequence is extended with the handling of state configuration changes, including regions, states (also considering history) and the corresponding entry and exit events (Phases 2-4 in Section 2.2). The mapping distinguishes *simple transitions* where both the source and target are stable states, and *complex transitions*, consisting of a sequence of transition elements connected by merge, join, fork and choice states.

Simple transitions In general, GSL and thus, the mapping follows the operational semantics of UML/SysML in terms of handling state configuration changes. Accordingly, the state configuration changes defined in Phase 3 of Section 2.2 are captured as follows.

- *Exiting* from states is mapped into an assign action that sets the variable corresponding to the parent region to contain the *inactive* literal or the corresponding *history literal* in case the region has history. *Exit actions* are also considered, procuring the creation of actions as described in Table 3.10.
- *Entering* to states is mapped into an assign action that sets the variable corresponding to the region to contain the corresponding state literal (*entry actions* are also handled according to Table 3.10). The new state configuration (considering history) is identified according to the rules defined by Phase 4 in Section 2.2.

The statements corresponding to exiting from states are inserted *before*, while the ones corresponding to state entries are inserted *after* the statements corresponding to the effects of the transition (in accordance with Phase 3 in Section 2.2).

Complex transitions In the case of complex transitions, join and merge, as well as fork and choice states are handled.

Table 3.10: Mapping of GSL statements into XSTS statements.

GSL statement	XSTS statement
Assignment	Assignment
Local variable	Local variable
Event raise	Assignment (set event boolean variable to true)
Block	Sequence
If-else	If-else
Switch	If-else
Choice	Choice
For loop	For loop
Function call	Body of the function (inline)

- Join states are mapped into *parallel* statements that contain the statements corresponding to the incoming transitions. The statements corresponding to outgoing transition of the join state are appended to the created parallel statement.
- Merge states are mapped into *choice* statements that contain the statements corresponding to the incoming transitions. The statements corresponding to outgoing transition of the merge state are appended to the created choice statement.
- Fork states are also mapped into *parallel* statements that contain the statements corresponding to the outgoing transitions. Note that, contrary to join states, the created parallel statement is inserted *after* the effect of the incoming transition (and does not contain it).
- Choice states are mapped into *choice* statements that contain the statements corresponding to the outgoing transitions. Similarly to fork states, the created statement is inserted *after* the effect of the incoming transition.

Transition composition according to SVP-* The *semantic variation points* (SVP-*) of GSL are considered when the created statements corresponding to the transitions of the statecharts are composed according to the rules of Phase 2 in Section 2.2 as follows:

- **SVP-2: conflict resolution between parent and child regions of hierarchical states** In case *top-down* conflict resolution is selected, the created statements corresponding to the transitions are composed into a sequence of *if-else* statements starting from the *topmost* region(s) and continuing with the nested ones, hierarchy level by hierarchy level (see the details of composition for a single hierarchy level in the following points). The conditions of the statements' assume actions (corresponding to the trigger and guard conditions of the particular region) are used as conditions of the *if-else* statements. In turn, in the case of *bottom-up* conflict resolution, the composition of *if-else* statements starts from the *lowermost* regions. However, a composite state may have multiple (orthogonal) child regions, each of which has a higher priority over the parent regions. Thus, in this case, a *local boolean variable* is introduced, which is set to true if any transition in the composite state's child regions fires; this variable is referenced in a negated form and connected in conjunction to the *if* condition corresponding to the parent's region, i.e., the transitions of the (parent) region may fire only if *none* of the nested regions' transition fired.
- **SVP-4: execution of orthogonal regions** As discussed above, the statements corresponding to the regions of transitions are composed hierarchy level by hierarchy level. In case a state has multiple child regions, the statements corresponding to the orthogonal regions' transitions are wrapped into either a *sequence*, *unordered* or *parallel* composite statement, corresponding to the *sequential*, *unordered* or *parallel* (see the *par* element at Line 24) execution of the orthogonal regions.
- **SVP-3: priority for transitions leaving the same state** The statements corresponding to a certain regions' transitions are composed according to this semantic variation point. In case *priorities* are defined, then a sequence of if-else statements (with the corresponding assume statements' conditions used as *if* conditions) are created in accordance with the transition priorities (see Lines 25-31). In case priorities are *absent*, then the corresponding statements are wrapped into a *choice* statement (see Lines 33-45).
- **SVP-1: evaluation of guard expressions** After the composition of the statements corresponding to the transitions, the "place" of evaluating guard expressions can be considered. In case *on-the-fly* evaluation is selected, then the emergent **trans** structure needs no modification (as

depicted in Figure 2.7). In case, *before execution* is selected, then the conditions of the corresponding if and assume statements are extracted into local boolean variables declared at the beginning of the **trans** structure, which are referenced from the original condition.

Model initialization The initialization of the model comprises two steps: (1) every variable is set to its initial value (default value of its type or initial expression of the variable declaration) and (2) entering the statechart, i.e., setting the variables corresponding to regions based on the initial states of the statechart while executing the corresponding entry actions of entered states. The created statement is used as the **init** transition of the XSTS model (see Lines 48-50).

Environment description By default, the mapping considers a fully *nondeterministic* environment, i.e., every boolean variable corresponding to an input event of the statechart is *havoc*ed, and in case the variable is true (*if* statement), the variables corresponding to the parameters of the event are also *havoc*ed (note that Lines 51-61 depict the handling of message queues of an asynchronous statechart, and not the input events directly). In addition, the variables corresponding to outputs events are set to false (as raised event values are valid for a single execution cycle) while setting the variables corresponding to its parameters to their default values in case the event is *transient* (thus, the values of *persistent* event parameters are retained). The created statement is used as the **env** transition of the XSTS model. Note that the framework supports the explicit definition of the environment as a GSL or GCL model; in this case, the **env** transition captures the behavior of this model.

In addition, the elapse of time is modeled by a special *delay* statement targeting the **clk** variables. Note that every clock measures time at the same rate.

3.4.2 Mapping composition constructs into XSTS

The mapping of composite components is conducted in a bottom-up way, starting from statecharts (synchronous components). According to the composition rules (see Section 3.3), first *synchronous* and *cascade* composite components are handled, which can be adapted to the asynchronous domain using *asynchronous adapters*, and potentially composed by *scheduled asynchronous* and *asynchronous* composite components.

Note that the model initialization and the environment description of these component models is analogous to that of statecharts, the difference is that now multiple components have to be considered and the events considered input and output from the system's perspective have to be identified based on *port bindings*.

Synchronous composite components In essence, the synchronous-reactive composition mode in XSTS can be modeled using an *orthogonal* statement, containing the corresponding **trans** transitions of the constituent synchronous components as operands. This way, it is ensured that signals produced by a component are sampled by other components in the next execution cycle only (see Definition 3.6). Regarding *channels*, the connection of outputs events to input events of the constituent components is modeled by introducing assign statements that (1) set the corresponding input event variable to true and (2) assign the parameter values to the corresponding input parameter variables. Recall that an orthogonal statement ensures the independent execution of the operands (corresponding to the execution of constituent components), i.e., event transmission does not occur during the execution of the orthogonal statement, only at the beginning of the next execution (see Section 2.4).

```
var eventQueue : [integer] -> integer // Storing event ids
var size : integer = 0; // Size variable

// Append
eventQueue[size] := eventId;
size := size + 1;
 // Peek
var peekVariable : integer = eventQueue[0];
... // Referencing peekVariable
// Pop
eventQueue[0] := eventQueue[1]; ...
eventQueue[size - 1] := 0; // Modeling an "empty cell"
size := size - 1;
```

Figure 3.9: *Append*, *peek* and *pop* message queue functions modeled in XSTS.

Cascade composite components The mapping of cascade composite components is similar to that of synchronous composite components, but as a difference, (1) a *sequence* is used instead of an *orthogonal* statement that contains as operands the **trans** transition of every component according to the specified execution sequence (note that it can reference the same particular component multiple times, see Definition 3.7) and (2) the **trans** transition is prepended with *assign* statements that *reset* the input event variables and in the case of *transient* events, also the input parameter variables. This way, in-cycle communication between components in a single execution cycle is ensured (see Definition 3.8).

Asynchronous adapter As their most important feature, asynchronous adapters introduce *message queues* to support the storage and serialization (one by one processing) of input event instances. In XSTS, a message queue is modeled as a set of *arrays*, depending on whether the queue has to store only non-parameterized events (*eventQueue* array, see Line 12) or also parameter values (one or more *argumentQueue* array for every *parameter type* to allow for storing each parameter value of each event – the number is determined by the maximum number of parameters of the same type for a single event; see Line 13) in addition to integer variables storing the number of contained elements (*size*, see Line 11). As a primary idea, every event type is assigned an integer identifier that is appended to the *eventQueue* array, modeling the append of the event instance to the corresponding message queue. In case the event is parameterized, the parameter value is appended to the corresponding *argumentQueue* array. Figure 3.9 summarizes how the supported *append*, *peek* and *pop* message queue functions are mapped into XSTS array handling constructs.

The adaptation of the referenced synchronous component is carried out in four steps.

1. An *if* statement is created that checks if any of the *eventQueue* arrays corresponding to the message queues is nonempty ($0 < size$ condition, see Line 15) and if so, it retrieves the stored event identifier of the highest priority nonempty queue (*peek* and *pop*, see Lines 16-17) and based on it, sets the corresponding input event variable to true (Line 19) and potentially, also loads the parameter values from the *argumentQueues* to the corresponding input parameter variables (*peek* and *pop*, Lines 20-21).
2. If the event initiates an execution (*trigger predicate*, see Definition 3.10), then the adapted component's **trans** function is wrapped into the created *if* statement; the following steps (Phases 3-4) are conducted only in this case.
3. The resulting transition is appended with an *unordered* sequence (see the discussion in regard to *event permutations* in Section 3.3.7) of *if* statements for every output event that checks if

```
if (outEvent) { // OutEvent is raised
 if (size >= CAPACITY) {
  // Pop
  eventQueue[0] := eventQueue[1]; ...
  eventQueue[size - 1] := 0;
  size := size - 1;
 }
 // Appending to the connected message queue
 eventQueue[size] := outEventId;
 size := size + 1;
}
```

```
if (outEvent && size < CAPACITY) {
 // Appending only if the connected message queue
    is not full
 eventQueue[size] := outEventId;
 size := size + 1;
}
```

Figure 3.10: XSTS construct for handling message transmission to message queues with the discard *oldest* and discard *incoming* message discard strategy.

the particular event is raised (corresponding event variable is set to true) and if so, appends the corresponding event identifier and parameter values to the *eventQueue* and *argumentQueue* arrays corresponding to every connected message queue (if any). Message queues support two discard strategies if they are full (see Section 3.3.1): discard the *incoming* message or the *oldest* one (at index 0) stored in the message queue – these strategies are also modeled in the second step with *if* statements and a potential *pop* function (see Figure 3.10).

4. Finally, the input and output event variables (and potentially the corresponding parameter variables in the case of *transient* events) are reset (i.e., the event vector is cleared) at the end (Line 46) and beginning (Line 23) of the **trans** transition, respectively.

The *clocks* of asynchronous adapters are mapped into clock variables, which are handled at the beginning of the **env** transition: if the value of the clock variable is greater than or equal to the specified rate, then it is reset and the *eventQueue* array corresponding to the message queue for storage is appended with a control event instance (i.e., its identifier).

Scheduled asynchronous composite component Scheduled asynchronous composite components simply wrap the **trans** transitions corresponding to the constituent asynchronous components into a *sequence* statement according to the specified execution sequence (see Definition 3.17).

Asynchronous composite component Asynchronous composite components wrap the **trans** transitions corresponding to the constituent asynchronous components into a *choice* statement. This way, an execution cycle corresponds to the execution of a *single* constituent asynchronous component (scheduled asynchronous or asynchronous adapter). As asynchronous components do not share internal states (e.g., shared variables), this mapping models every possible behavior of the asynchronous composite component apart from the potential *interleaving* of messages sent by components running in parallel (see Section 3.3.8.2). Thus, at the beginning of **trans**, the arrays corresponding to the components' message queues are saved to store the original order of messages, which are then used to create every possible permutation of messages (original and newly sent) at the end of **trans**. Note that this step may greatly increase the state space of the model.

3.5 Component integration and verification workflow

This section overviews the general workflow according to which components can be integrated and verified in the Gamma framework, incorporating the modeling languages and formal verification

functionalities presented in previous sections. This workflow also serves as a foundation for the integration of high-level state-based components[4] and their formal verification, which can be reused and extended to support additional design and verification functionalities, e.g., test generation as presented in Chapter 4 and contract-based verification as presented in Chapter 5. We introduce the workflow in the context of the simplified *elevator system* presented in Section 2.3. The workflow is built on top of the model transformation chain depicted in Figure 3.1. In the following, we overview the used modeling languages.

- The **Gamma Statechart Language (GSL)** is a UML/SysML-based statechart language supporting different semantic variants of statecharts (presented in Section 3.2).
- The **Gamma Composition Language (GCL)** is a composition language for the formal hierarchical composition of state-based components according to multiple execution and interaction semantics (presented in Section 3.3).
- The **Gamma Genmodel Language (GGL)** is a configuration language for configuring model transformations in the framework.
- The **Gamma Property Language (GPL)** is a property language supporting the definition of CTL* [EH86] properties and thus, the formal specification of requirements regarding (composite) component behavior.
- The **Gamma Trace Language (GTL)** is a high-level description language for execution traces of (composite) components with synchronous and asynchronous communication (to be presented in detail in Section 4.1).

The following sections outline the consecutive steps of the general component integration and verification workflow in the Gamma framework. As the first (optional) step in the workflow, statechart models defined in integrated modeling tools (front-ends) are imported into Gamma, which can be hierarchically integrated using well-defined composition modes (see Section 3.5.1). The emergent composite model is processed and transformed into the input formalisms of integrated model checker back-ends for formal verification (Section 3.5.2). The back-ends provide witnesses (diagnostic traces) based on specified properties, which are back-annotated into abstract execution traces (Section 3.5.3)

3.5.1 Creating functional component and system designs

Optionally, the import of an external component model, i.e., a statechart created in an external modeling tool (currently Yakindu, MagicDraw and SCXML are supported), is realized by executing a model transformation that maps this model into a GSL statechart (see Section 3.2 and [*e*12], as well as [Gra16] for the integration of Yakindu). Alternatively or complementarily, statechart models can be defined directly in GSL. GSL serves as a common representation language for component statecharts with different operational semantics configurable by *annotations* (semantic variation points, see Section 3.2). Validation also takes place on Gamma statecharts by evaluating well-formedness constraints.

The GSL models generated from the *cabin controller* and *cabin door controller* Yakindu models are presented in a textual format in Figures C.1 and C.2 in the Appendix, respectively.

Next, GSL models can be hierarchically integrated according to various precise execution and interaction semantics in GCL to create *synchronous* or *asynchronous* systems using the *synchronous-reactive*, *cascade*, *asynchronous-reactive* and *scheduled asynchronous-reactive* composition modes (see Section 3.3); the resulting composite models are also validated against well-formedness constraints.

The GSL models of our elevator example can be integrated in GCL to create composite models conforming to different composition semantics as presented in Figure C.3.

[4]Even though, the Gamma framework currently supports statechart models as input, its modeling language family supports the integration of other formalisms, e.g., there is ongoing work on integrating activity diagrams [Zav21].

3.5.2 Processing composite models

Model checkers can carry out exhaustive analysis on a formal (analysis) model based on a formal property and determine if the property holds while potentially providing a diagnostic trace (counterexample or witness) as proof [CHV18]. Gamma facilitates deriving both the analysis model (see Section 3.4) and the property using automated semantic-preserving model transformations.

In order to derive analysis models, the composite models (see Section 3.5.1) are preprocessed and transformed into the input languages of integrated model checker back-ends (see Section 3.4). The transformation can be configured in GGL in a textual format, e.g., the following snippet specifies that the *Elevator* composite model is to be transformed into Promela (input formalism of Spin).

```
component : Elevator
language : Promela
```

Gamma facilitates the specification of CTL* properties in GPL using a textual syntax. GPL supports referencing certain elements of the composite model, i.e., *states*, *variables*, *events* and *event parameters*, as well as *transitions* annotated with an identifier. Note that whether the specified properties can actually be checked depends on the selected model checker back-end as most model checkers support only a subset of CTL* as an input property language.

In the elevator example, we can specify two properties for the *cabin door controller* component to check the execution of the transitions between the *Open* and *Closed* and the *Closed* and *Open* states. Note that a transition can be referred to in GPL if it is assigned an annotation specifying an identifier (see *Transition* entry in Table 3.2).

```
@("Covering the transition going from state 'Open' to 'Closed' in the 'main' region")
E F (var main_open__main_closed)
@("Covering the transition going from state Closed' to 'Open' in the 'main' region")
E F (var main_closed__main_open)
```

Both the composite model and the properties are automatically transformed into the input languages of the selected model checker back-ends in accordance with the internal components and the characteristics of the used composition modes. In order to reduce the state space of the model, the transformations exploit optimization techniques based on variables that store *resettable* or *transient* data (see Section 3.2).

The transformation of GPL properties into the property languages of model checkers is carried out based on the traceability links defined in the composite model transformation, as in this context, only the identifiers of the target model elements are required.

3.5.3 Formal verification and back-annotation

As a key feature, Gamma offers multiple model checker back-ends to complement each other and facilitate efficient verification of different models and properties. Currently, UPPAAL, Theta and Spin are integrated as back-ends, but the integration of additional model checkers is allowed by the framework, e.g., by building on the DX transformation (see Section 3.4).

Gamma supports the tuning of the model checking process in GGL, e.g., the specification of search strategies, such as breadth-first search (BFS) or depth-first search (DFS), or the selection of abstraction techniques. The framework also provides generally well-functioning default settings for the back-ends. Model checking itself is viewed as a black-box process with the generated analysis models and properties as inputs and diagnostic traces as outputs. Diagnostic traces specify the steps (active states and output events of the model in response to input events coming from the environment) that lead

to the satisfaction of a certain property in case the property is satisfiable. Model checkers can also prove that certain properties are impossible to satisfy, which can be essential information during verification. The diagnostic traces are automatically back-annotated and abstract execution traces are created in GTL (see Section 4.1) in a textual format[5] based on the traceability links defined in the corresponding composite model transformation.

As an example, the snippets in Figures 4.2 and 4.3 describe GTL execution traces derived on the basis of the `E F (var main_open__main_closed)` and `E F (var main_closed__main_open)` GPL properties defined in the previous section.

3.6 Case study – Orion protocol

This section demonstrates a formal analysis approach for communication protocols using the Gamma framework. 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) formal verification 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.[6]

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.

3.6.1 Modeling of the communication protocol

This section presents the *modeling process* of the proposed analysis approach in the context of Orion, including the modeling of protocol participants (Section 3.6.1.1), channel models (Section 3.6.1.2) and the integrated system models (Section 3.6.1.3).

3.6.1.1 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 on the left of Figure 3.11) 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*. The *slave* statechart (depicted on the right of Figure 3.11) is similar to the master.

The models can be automatically transformed into GSL using the model transformers of Gamma, and validated based on statechart-related well-formedness rules [Gra16]. According to the model validators of the framework, the presented statechart models are well-formed.

[5] The framework also supports the visualization of GTL execution traces using PlantUML.

[6] Every Gamma model presented in this case study can be found at https://github.com/ftsrg/gamma/tree/v2.9.0/examples/hu.bme.mit.gamma.industrial.protocol.casestudy.

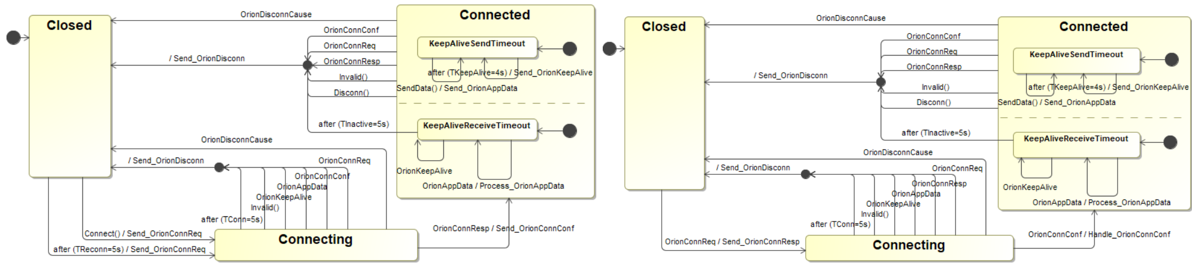

Figure 3.11: Statechart models describing the behavior of the *master* and *slave* components.

3.6.1.2 Channel models

During the modeling of communication between protocol participants, several failure modes of event transmission can be considered [PF20]. 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 [*e*14] in detail, here we present only the simplified version of the *bursty message losing channel* model.

Figure 3.12 depicts an excerpt of the *bursty message losing channel* model, which models a channel that can lose a given amount (*LOST_MESSAGE_MAX*) of subsequent incoming events. The model 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) enters state *MessageLosing* without forwarding it. In state *MessageLosing*, the specified amount of events are absorbed before going back to state *Operating*. Note the nondeterministic nature of this model: the loss of subsequent events can start upon any incoming event. Moreover, this model excerpt includes behavior only for a single event (*OrionConnReq*); nonetheless, additional events in the complete model are handled analogously.

3.6.1.3 System models

We analyzed the behavior of the Orion protocol considering different channel failure modes and different composition modes of the participants. Therefore, for each channel model we defined *synchronous*, *cascade* and *asynchronous* composite Gamma models, which differ only in the composition 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.

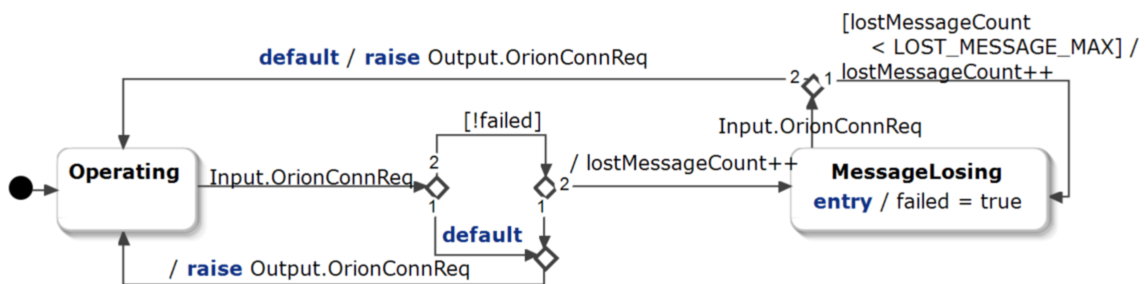

Figure 3.12: Excerpt from the statechart model of the *bursty message losing channel*.

```
[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]
}
```

Figure 3.13: GCL model variants of protocol participants and channel models.

Figure 3.13 describes the GCL model the variations of which were used with different channel models and composition 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 two were used.

3.6.2 Analysis of the communication protocol

We analyzed liveness properties of the system models, that is, the reachability of system states with different channel models using the integrated UPPAAL model checker via the DU transformation (see Section 3.4). The analyzed properties (formalized in CTL [CE81]) are the following:

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 (examined from the initial state) is always able to recover despite the channel's specified failure mode.

According to the verification executed in Gamma, P_1 *holds* in the case of *every system model*. 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 satisfaction 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 held in every system model with adequate execution characteristics. The detailed constraints on the execution characteristics to satisfy the property can be found in [*e*14].

In order to provide additional insight into the verification capability of Gamma, we measured the time (see Table 3.11) and the memory consumption (see Table 3.12) 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

Table 3.11: Average time of verifying P_2 in different system models.

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

was 1. The execution frequencies were set in accordance with the constraints mentioned above. We ran and averaged 10 measurements for each system model.

3.6.3 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 used. 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 remarkable in the case of the *arbitrary message losing channel* where there is a 70-fold difference in verification time compared to the synchronous model.

3.6.4 Conclusion

This case study demonstrated that the composition modes proposed by Gamma can indeed be used in practice: even though the semantics introduced in Section 3.3 assumes instantaneous, reliable and order-preserving message passing, the language is applicable to practical problems by introducing (fault) models describing physical phenomena, e.g., the loss or delay of events.

The three applied composition modes 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

Table 3.12: Average resident/virtual memory peaks during the verification of P_2 in different system models.

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

Table 3.13: Features of the Gamma framework and its “competitors.” ✓ = full support; ✓̸ = experimental

	Native statecharts	Mixed-semantic composition	Formal semantics	Synchronization-based	Event-based	Stoc. process algebra 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	✓		✓		✓		✓		✓	

the effort required for the approach. Also, the measurement results show that the formal verification capabilities of the framework are applicable in practice.

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

3.7 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 [Szt+14; Sim+13], 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 [Miy+19] 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 [Jia+18] 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 [Juo+18] 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 [Lug+16; AR16] 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 [HRR14; Chi+06; FGH06; Rei17; Bro97] capture the architecture, mainly focusing on the interfaces, connectors and their relations in systems without defining the behavior of the components. In [GBC10], 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 3.13: 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.

In [Chh+15], a modeling environment is introduced that supports the graphical definition of SystemC [Pan01] 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 between the module and its environment based on events declared by interfaces. Ports can be connected by channels. The environment supports 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 [Her10]. STATE maps the informal SystemC code to a formal timed automaton formalism, UPPAAL, thus provides formal verification capabilities. Contrary to Gamma, this modeling environment currently does not support the hierarchical and mixed-semantic composition of modules.

Æmilia [BDC02] 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 stochastic behavior.[7] 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, mixed-semantic composition and related code generation.

CHESS [Maz+16] 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 control 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

[7] However, an extension to Gamma introducing stochastic behavior is under development [*c*7].

is responsible for the interactions 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 mixed-semantic composition of components.

BIP [BBS06] (Behavior, Interaction, Priority) is a modeling framework supporting the formal definition of heterogeneous systems. The BIP language supports the definition of hierarchical composite systems in 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 are based on a rigorous transition system model, thus, formal verification of invariant properties and deadlock-freedom is also supported.

Ptolemy II [Eke+03] 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 [Bae+12].

The UML-RT [Sel98] is an UML profile (evolved from the ROOM language [SGW94]) used by IBM Rational Software Architect RealTime Edition (RSA RTE) and alternatively by Papyrus RT[HDB17]. 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 an 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 [HDB17]. UML-RT models can be transformed to a rigorous state machine formalism, called CFFSM [Zur14; ZD17], which can be formally analyzed by their model checker based on CTL expressions. Nevertheless, this tool does not fully support mixed-semantic composition of components. Test generation is also supported in [RD12]. 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.

In [Ara+15] a model-based tool and research platform for safety-critical embedded systems is introduced. AutoFOCUS 3 (AF) 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 [BS12]. AF (similarly to Gamma) employs a

multi-level transformation chain to produce formal models from high level design models. Moreover, it has a modeling language based on the statechart formalism and the tool provides editor support, too. As opposed to AF, our approach promotes to use a common formal representation (GSL) and integrate the models of the different design tools: models developed in the AF tool could also serve as input for Gamma. Consequently, Gamma could serve as an integration tool for AF models. AF provides verification mainly for synchronously (according to weak or strong causality) composed software models [KA17] as it is integrated with well-known symbolic model checkers, such as NuSMV/nuXmv, which provide efficient verification capabilities for synchronous systems. Contrarily, Gamma uses UPPAAL, Theta and Spin for model checking, which feature different algorithms (explicit-state and symbolic) and are specialized to systems with different characteristics and use-cases.

xtUML (eXecutable and Translatable UML) [SM88] is an UML-based modeling language with a special focus on executable semantics. State machines with an expressive action language are used to define component behavior. 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.

COMDES-II (COMponent-based design of software for Distributed Embedded Systems) [KSA07] is a design tool for layered component models where synchronous and asynchronous behavior is 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 [Ke+08] is based on the UPPAAL model checker.

ProCom [Bur+08] is a component model for real-time embedded systems that employs a layered component model comprising two distinct, but related layers. At the upper layer (called ProSys) the system is modeled as a number of active and concurrent subsystems, communicating 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 [Vul+09; BC11] that focuses on reactive and real-time aspects and supports the co-existence of black-box and fully implemented system components.

3.8 Summary and future work

This chapter introduced the Gamma Statechart Composition framework, a modeling framework for the component-based design and forma verification of reactive systems. It presented the framework's statechart language (GSL) featuring semantic variation points to support capturing the behaviors of different statechart variants (operational semantics). It also formally presented the composition language of the framework (GCL) to support the description of synchronous behavior with the synchronous-reactive and cascade, as well as asynchronous behavior with the asynchronous-reactive and scheduled asynchronous-reactive composition modes. Model transformations giving denotational semantics to the GSL and GCL languages, as well as supporting the formal verification of integrated statechart models by mapping them into the inputs of integrated model checker back-ends were also presented.

The chapter also featured a case study, which demonstrated that the expressiveness and usability of the introduced modeling languages are adequate to capture the functional behavior of real-life component-based reactive systems. In particular, the case study also showed how the functional models can be extended to refine the ideal characteristics of message passing considered in the composition language to analyze the functional behavior of protocols in real-life scenarios. The feasibility of

the framework's formal verification capabilities, considering different composition modes, was also demonstrated.

The contributions of this chapter were the following:

Contribution group 1 I developed a *modeling framework* for the *component-based design* and *formal exhaustive verification* of *composite reactive systems*, which consists of the following contributions:

1. I developed a *statechart language* with formal semantics featuring *semantic variation points* to support different internal *execution modes* of statecharts (i.e., statechart variants). Accordingly, the statechart language allows for the *extension* of the framework with additional modeling front-ends [*j*1].
2. I developed a *composition language* with formal semantics that supports the hierarchical *mixed-semantics composition* of heterogeneous statechart components according to various well-defined (formal) execution and interaction semantics: asynchronous-reactive, scheduled asynchronous-reactive, synchronous-reactive, and cascade [*e*12][*e*13][*j*1].
3. I developed semantic-preserving automated *model transformations* that map the high-level composite models into the input formalisms of model checker back-ends, namely, timed automata, transition systems and process models. They support the *exhaustive formal verification* of composite models, providing flexibility for time-dependent, data-oriented or parallel behavior, as well as the automated back-annotation of the results to the source composite models [*j*1][*j*4][*c*8].
4. I developed an *extensible workflow* (component integration and verification workflow) that incorporates modeling front-ends, the statechart language, the composition language and the model transformations and back-annotation facilities of the framework to allow for the black box (but also customizable) application of formal verification (model checking) on high-level integrated state-based models as a *reusable functionality* [*j*1][*j*4][*c*6][*c*8].

The framework's statechart language (Contribution 1.1) was discussed in Section 3.2, whereas the composition language (Contribution 1.2) was presented in Section 3.3. The model transformations that allow for the formal verification of integrated models (Contribution 1.3) were introduced in Section 3.4. The component integration and verification workflow (Contribution 1.4) was presented in Section 3.5.

Subject to future work, we aim to integrate additional model checkers to the framework, for instance, nuXmv [BB14; Cav+14], to provide even greater flexibility for verifying models with different characteristics. We also plan to support the custom definition of composition (execution and interaction) modes for component integration to aid engineers in experimenting with different composition semantics during system design. In addition, we are working on integrating SysML v2 [Gro20] to the framework to aid industrial parties in the semantically sound composition and verification of system models.

Publications

Publications related to the contributions

	Journal papers	International conference papers	Local events
Contribution group 1	[*j*1],[*j*4]	[*c*6],[*c*7],[*c*8],[*c*10]	[*e*12], [*e*13], [*e*14], [*e*15]

Journal papers

[*j*1] **Bence Graics**, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling* 19, 2020, pp. 1483–1517. DOI: 10.1007/s10270-020-00806-5.

[*j*2] **Bence Graics**, Vince Molnár, and István Majzik. Integration test generation for state-based components in the Gamma framework, 2022.

[*j*3] **Bence Graics**, Vince Molnár, and István Majzik. Component-based specification, design and verification of adaptive systems. *Systems Engineering* n/a(n/a), 2023. DOI: 10.1002/sys.21675.

[*j*4] Benedek Horváth, Vince Molnár, **Bence Graics**, Ákos Hajdu, István Ráth, Ákos Horváth, Robert Karban, Gelys Trancho, and Zoltán Micskei. Pragmatic verification and validation of industrial executable SysML models. *Systems Engineering*, 2023. DOI: 10.1002/sys.21679.

[*j*5] **Bence Graics**, Milán Mondok, Vince Molnár, and István Majzik. Test generation and formal verification for asynchronously communicating distributed controllers, 2023.

International conference papers

[*c*6] Vince Molnár, **Bence Graics**, András Vörös, István Majzik, and Dániel Varró. The Gamma Statechart Composition Framework. In: *40th International Conference on Software Engineering (ICSE)*, pp. 113–116. Gothenburg, Sweden: ACM, 2018. DOI: 10.1145/3183440.3183489.

[*c*7] Simon József Nagy, **Bence Graics**, Marussy Kristóf, and András Vörös. Simulation-based safety assessment of high-level reliability models. In: *4th Workshop on Models for Formal Analysis of Real Systems*, pp. 240–260. 2020. DOI: 10.4204/EPTCS.316.9.

[*c*8] Benedek Horváth, **Bence Graics**, Ákos Hajdu, Zoltán Micskei, Vince Molnár, István Ráth, Luigi Andolfato, Ivan Gomes, and Robert Karban. Model checking as a service: towards pragmatic

hidden formal methods. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pp. 1–5. 2020. DOI: 10.1145/3417990.3421407.

[c9] **Bence Graics**, Vince Molnár, and István Majzik. Contract-based specification and test generation for adaptive systems. In: *16th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX)*, Advances in Intelligent Systems and Computing, vol. 1389, pp. 136–145. Wrocław, Poland: Springer, 2021. DOI: 10.1007/978-3-030-76773-0.

[c10] Danilo Pallamin de Almeida, **Bence Graics**, Ronan Arraes Jardim Chagas, Fabiano Luis de Sousa, and Fatima Mattiello-Francisco. Towards simulation of CubeSat operational scenarios under a cyber-physical systems view. In: *10th Latin-American Symposium on Dependable Computing (LADC 2021)*, Florianópolis, Brazil: IEEE, 2021. DOI: 10.1109/LADC53747.2021.9672594.

[c11] **Bence Graics**, Milán Mondok, Vince Molnár, and István Majzik. Configurable model-based test generation for distributed controllers using declarative model queries and model checkers. In: 2023.

Local conference papers

[e12] **Bence Graics** and Vince Molnár. Formal compositional semantics for Yakindu statecharts. In: Béla Pataki (ed.), *24th Minisymposium of the Department of Measurement and Information Systems*, pp. 22–25. Budapest, Hungary, 2017.

[e13] **Bence Graics** and Vince Molnár. Mix-and-match composition in the Gamma framework. In: Béla Pataki (ed.), *25th Minisymposium of the Department of Measurement and Information Systems*, pp. 24–27. Budapest, Hungary, 2018.

[e14] **Bence Graics** and István Majzik. Modeling and analysis of an industrial communication protocol in the Gamma framework. In: Balázs Renczes (ed.), *27th Minisymposium of the Department of Measurement and Information Systems*, pp. 25–28. Budapest, Hungary, 2020.

[e15] Csanád Csuvarszki, **Bence Graics**, and András Vörös. Model-driven development of heterogeneous cyber-physical systems. In: Balázs Renczes (ed.), *28th Minisymposium of the Department of Measurement and Information Systems*, Budapest, Hungary, 2021.

[e16] **Bence Graics** and István Majzik. Integration test generation and formal verification for distributed controllers. In: Balázs Renczes (ed.), *30th Minisymposium of the Department of Measurement and Information Systems*, pp. 1–4. Budapest, Hungary, 2023. DOI: 10.3311/minisy2023-001.

Bibliography

[Adl+07] Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. In: Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie (eds.), *Formal Methods and Software Engineering*, pp. 76–95. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-76650-6_6.

[Adl+11] Rasmus Adler, Ina Schaefer, Mario Trapp, and Arnd Poetzsch-Heffter. Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. *ACM Trans. Embed. Comput. Syst.* 10(2), 2011. DOI: 10.1145/1880050.1880056.

[AR16] L. Apvrille and Y. Roudier. Model-driven engineering and software development. In: Springer International Publishing, 2016. Chap. Designing Safe and Secure Embedded and Cyber-Physical Systems with SysML-Sec, pp. 293–308.

[Ara+15] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: tooling concepts for seamless, model-based development of embedded systems. *ACES-MB&WUCOR@ MoDELS* 1508, 2015, pp. 19–26.

[Bae+12] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, Edward A. Lee, and Stavros Tripakis. Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Science of Computer Programming* 77(12), 2012, pp. 1235–1271. DOI: https://doi.org/10.1016/j.scico.2010.10.002.

[Bas+11] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. 28, 2011, pp. 41–48.

[BB14] Armin Biere and Roderick Bloem, eds. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings.* Vol. 8559. Lecture Notes in Computer Science. Springer, 2014. DOI: 10.1007/978-3-319-08867-9.

[BB91] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE* 79(9), 1991, pp. 1270–1282.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In: *Software Engineering and Formal Methods (SEFM) 2006. Fourth IEEE International Conference*, pp. 3–12. 2006.

[BC11] Etienne Borde and Jan Carlson. 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.

[BC14] Jung Ho Bae and Heung Seok Chae. Systematic approach for constructing an understandable state machine from a contract-based specification: controlled experiments. *Software & Systems Modeling*, 2014, pp. 1–33.

[BDC02] Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. Stochastic process algebra: from an algebraic formalism to an architectural description language. In: *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pp. 236–260. 2002.

[Beh+06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. IEEE Computer Society, 2006, pp. 125–126.

[Ben+16] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. Component-based verification using incremental design and invariants. *Software and System Modeling* 15(2), 2016, pp. 427–451.

[BFS05] Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for test case generation. In: *Model-Based Testing of Reactive Systems*, pp. 391–438. Springer, 2005.

[BK11] Björn Bartels and Moritz Kleine. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pp. 158–167. Association for Computing Machinery, 2011. doi: 10.1145/1988008.1988030.

[BRG08] Philipp A Baer, Roland Reichle, and Kurt Geihs. The SPICA development framework-model-driven software development for autonomous mobile robots. In: *Intelligent Autonomous Systems 10*, pp. 211–220. IOS Press, 2008.

[Bri+04] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Live Sequence Charts. In: *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report.* Ed. by Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper. Springer Berlin Heidelberg, 2004, pp. 374–399. doi: 10.1007/978-3-540-27863-4_21.

[Bro97] Manfred Broy. Compositional refinement of interactive systems. *J. ACM* 44(6), 1997, pp. 850–891. doi: 10.1145/268999.269004.

[BS12] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer Science & Business Media, 2012.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In: *Handbook of model checking*, pp. 305–343. Springer, 2018.

[Bur+08] Tomáš Bureš, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the progress component model reference manual. *Mälardalen University, Västerås, Sweden*, 2008.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: semantics, algorithms and tools. In: pp. 87–124. Springer-Verlag, 2004.

[Cal+18] Radu Calinescu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli, and Tim Kelly. Engineering trustworthy self-adaptive software with dynamic assurance cases. *IEEE Transactions on Software Engineering* 44(11), 2018, pp. 1039–1069. DOI: 10.1109/TSE.2017.2738640.

[Cav+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In: Armin Biere and Roderick Bloem (eds.), *CAV*, Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_22.

[CD05a] Michelle L. Crane and Juergen Dingel. On the semantics of UML state machines: Categorization and comparision. In: *In Technical Report 2005-501, School of Computing, Queen's*, 2005.

[CD05b] Michelle L. Crane and Juergen Dingel. UML vs. classical vs. Rhapsody statecharts: Not all models are created equal. In: Lionel Briand and Clay Williams (eds.), *Model Driven Engineering Languages and Systems*, pp. 97–112. Springer, 2005.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In: *Workshop on Logic of Programs*, pp. 52–71. 1981.

[Cha+19] Ronan AJ Chagas, Fabiano L de Sousa, Arcélio C Louro, and Willer G dos Santos. Modeling and design of a multidisciplinary simulator of the concept of operations for space mission pre-phase A studies. *Concurrent Engineering* 27(1), 2019, pp. 28–39. DOI: 10.1177/1063293X18804006.

[Che+09] Betty H. C. Cheng et al. Software engineering for self-adaptive systems: a research roadmap. In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Springer Berlin Heidelberg, 2009, pp. 1–26. DOI: 10.1007/978-3-642-02161-9_1.

[Chh+15] Ajay Chhokra, Sherif Abdelwahed, Abhishek Dubey, Sandeep Neema, and Gabor Karsai. From system modeling to formal verification. *The 2015 Electronic System Level Synthesis Conference*, 2015.

[Chi+06] Adam Childs, Jesse Greenwald, Georg Jung, Matthew Hoosier, and John Hatcliff. Calm and Cadena: metamodeling for component-based product-line development. *Computer* 39(2), 2006, pp. 42–50.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* SE-4(3), 1978, pp. 178–187. DOI: 10.1109/TSE.1978.231496.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to model checking. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 1–26. DOI: 10.1007/978-3-319-10575-8_1.

[CL12] Javier Cámara and Rogério de Lemos. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 53–62. 2012. DOI: 10.1109/SEAMS.2012.6224391.

[Cla+18] Edmund M. Clarke Jr., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[CLS00] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In: Mogens Nielsen and Dan Simpson (eds.), *Application and Theory of Petri Nets 2000*, pp. 103–122. Springer, 2000. doi: 10.1007/3-540-44988-4_8.

[CS03] Gianfranco Ciardo and Radu Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In: Warren A. Hunt and Fabio Somenzi (eds.), *Computer Aided Verification*, pp. 40–53. Springer, 2003. doi: 10.1007/978-3-540-45069-6_4.

[Czi+17] Bence Czipó, Ákos Hajdu, Tamás Tóth, and István Majzik. Exploiting hierarchy in the abstraction-based verification of statecharts using SMT solvers. *Electronic Proceedings in Theoretical Computer Science* 245, 2017, pp. 31–45. doi: 10.4204/eptcs.245.3.

[DAC99] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. 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. doi: 10.1145/302405.302672.

[DH01] Werner Damm and David Harel. LSCs: breathing life into Message Sequence Charts. *Formal Methods in System Design* 19(1), 2001, pp. 45–80. doi: 10.1023/A:1011227529550.

[Dor+05] R. Dorofeeva, K. El-Fakih, S. Maag, A.R. Cavalli, and N. Yevtushenko. Experimental evaluation of FSM-based testing methods. In: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pp. 23–32. 2005. doi: 10.1109/SEFM.2005.17.

[Ebn07] Ali Ebnenasir. Designing run-time fault-tolerance using dynamic updates. In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*, pp. 15–15. 2007. doi: 10.1109/SEAMS.2007.5.

[EH86] E. Allen Emerson and Joseph Y. Halpern. ''Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM* 33(1), 1986, pp. 151–178. doi: 10.1145/4904.4999.

[Eke+03] Johan Eker, Jorn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE* 91(1), 2003, pp. 127–144. url: http://chess.eecs.berkeley.edu/pubs/488.html.

[EL03] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming* 48(1), 2003, pp. 21–42. doi: https://doi.org/10.1016/S0167-6423(02)00096-5.

[Eno+16] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer* 18(3), 2016, pp. 335–353.

[Esh09] Rik Eshuis. Reconciling statechart semantics. *Science of Computer Programming* 74(3), 2009, pp. 65–99.

[FGH06] Peter H Feiler, David P Gluch, and John J Hudak. *The architecture analysis & design language (AADL): An introduction*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[Flo+10] J. Floch, C. Carrez, P. Cieślak, M. Rój, R.T. Sanders, and M.M. Shiaa. A comprehensive engineering framework for guaranteeing component compatibility. *Journal of Systems and Software* 83(10), 2010, pp. 1759–1779. DOI: https://doi.org/10.1016/j.jss.2010.04.075.

[Fox11] Jorge Fox. A formal orchestration model for dynamically adaptable services with COWS. In: *International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2011.

[FWA09] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability* 19(3), 2009, pp. 215–261. DOI: 10.1002/stvr.402.

[GBC10] Sebastien Gerard, Jean-Philippe Babau, and Joel Champeau. *Model Driven Engineering for Distributed Real-Time Embedded Systems*. Wiley-IEEE Press, 2010.

[Gei13] Kurt Geihs. Self-adaptivity from different application perspectives. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Springer Berlin Heidelberg, 2013, pp. 376–392. DOI: 10.1007/978-3-642-35813-5_15.

[GLM18] Hubert Garavel, Frédéric Lang, and Laurent Mounier. Compositional verification in action. In: Falk Howar and Jiří Barnat (eds.), *Formal Methods for Industrial Critical Systems*, pp. 189–210. Springer International Publishing, 2018.

[Gra16] Bence Graics. *Documentation of the Gamma Statechart Composition Framework v0.9*. Tech. rep. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2016. URL: https://tinyurl.com/yeywrkd6.

[Gra18] Bence Graics. *Documentation of the Gamma Statechart Composition Framework v2.0*. Tech. rep. https://tinyurl.com/2xxyujtf. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2018.

[Gro12] Object Management Group. *Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 2: Superstructure*. Tech. rep. ISO/IEC 19505-2:2012. Object Management Group, 2012. URL: http://www.omg.org/spec/UML/ISO/19505-2/PDF/.

[Gro18] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models*. Tech. rep. formal/2018-12-01. Object Management Group, 2018. URL: https://www.omg.org/spec/FUML/1.4/PDF.

[Gro19a] Object Management Group. *Precise Semantics of UML Composite Structures (PSCS)*. Tech. rep. formal/2019-02-01. Object Management Group, 2019. URL: https://www.omg.org/spec/PSCS/1.2/PDF.

[Gro19b] Object Management Group. *Precise Semantics of UML State Machines (PSSM)*. Tech. rep. formal/2019-05-01. Object Management Group, 2019. URL: https://www.omg.org/spec/PSSM/1.0/PDF.

[Gro20] Object Management Group. *Systems Modeling Language Version 2 (SysML v2)*. Standard. Object Management Group (OMG), 2020. URL: http://www.omgsysml.org/.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In: *Proceedings of the First Workshop on Self-Healing Systems*, WOSS '02, pp. 27–32. Association for Computing Machinery, 2002. DOI: 10.1145/582128.582134.

[GT18] Havva Gulay Gurbuz and Bedir Tekinerdogan. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal* 26(4), 2018, pp. 1327–1372. doi: 10.1007/s11219-017-9386-2.

[Haj+16] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In: Elvira Albert and Ivan Lanese (eds.), *Formal Techniques for Distributed Objects, Components and Systems*, Lecture Notes in Computer Science, vol. 9688, pp. 158–174. Springer, 2016. doi: 10.1007/978-3-319-39570-8_11.

[Hal+91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE* 79(9), 1991, pp. 1305–1320.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* 8(3), 1987, pp. 231–274. doi: 10.1016/0167-6423(87)90035-9.

[HDB17] N. Hili, J. Dingel, and A. Beaulieu. 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.

[Hea+09] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A case study in goal-driven architectural adaptation. In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Springer Berlin Heidelberg, 2009, pp. 109–127. doi: 10.1007/978-3-642-02161-9_6.

[Hea63] B. R. Heap. Permutations by Interchanges. *The Computer Journal* 6(3), 1963, pp. 293–298. doi: 10.1093/comjnl/6.3.293.

[Heg+10] Ábel Hegedüs, Gábor Bergmann, István Ráth, and Dániel Varró. Back-annotation of simulation traces with change-driven model transformations. *Proceedings of the Software Engineering and Formal Methods (SEFM) 2010*, 2010, pp. 145–155. doi: 10.1109/SEFM.2010.28.

[Hei+04] Mats P. E. Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: a case study. In: Alexandre Petrenko and Andreas Ulrich (eds.), *Formal Approaches to Software Testing*, pp. 42–59. Springer, 2004. doi: 10.1007/978-3-540-24617-6_4.

[Her10] Paula Herber. *A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata*. Logos Verlag Berlin GmbH, 2010.

[HLS01] Hyoung Hong, Insup Lee, and Oleg Sokolsky. Automatic test generation from statecharts using model checking. *Technical Reports (CIS)*, 2001.

[HM00] Thomas A. Henzinger and Rupak Majumdar. A classification of symbolic transition systems. In: Horst Reichel and Sophie Tison (eds.), *STACS 2000*, pp. 13–34. Springer Berlin Heidelberg, 2000.

[HMR] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In: pp. 261–270. doi: 10.1109/SEFM.2004.1347530.

[HN04] Alan Hartman and Kenneth Nagin. The AGEDIS tools for model based testing. *ACM Sigsoft Software Engineering Notes* 29, 2004. doi: 10.1145/1007512.1007529.

[Hoa21] C.A.R. Hoare. Communicating sequential processes. In: *Theories of Programming: The Life and Works of Tony Hoare*. 1st ed. Association for Computing Machinery, 2021, pp. 157–186. doi: 10.1145/3477355.3477364.

[Hol05] Gerard J. Holzmann. Software model checking with Spin. In: Advances in Computers, vol. 65, pp. 77–108. Elsevier, 2005. doi: https://doi.org/10.1016/S0065-2458(05)65002-4.

[Hol11] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* 1st. Addison-Wesley Professional, 2011.

[HRR14] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - architectural modeling of interactive distributed and cyber-physical systems. *ArXiv* abs/1409.6578, 2014.

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* 1st. Addison-Wesley Longman Publishing Co., Inc., 1990.

[IW14] M. Usman Iftikhar and Danny Weyns. ActivFORMS: active formal models for self-adaptation. In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pp. 125–134. Association for Computing Machinery, 2014. doi: 10.1145/2593929.2593944.

[Jia+18] Yu Jiang, Houbing Song, Yixiao Yang, Han Liu, Ming Gu, Yong Guan, Jiaguang Sun, and Lui Sha. Dependable model-driven development of CPS: from stateflow simulation to verified implementation. *ACM Trans. Cyber-Phys. Syst.* 3(1), 2018, 12:1–12:31. doi: 10.1145/3078623.

[JM99] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In: *International Conference on Computer Aided Verification*, pp. 108–122. 1999.

[Juo+18] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: lightweight modeling of structure, behaviour, and variability. *The Art, Science, and Engineering of Programming* 3(1), 2018, pp. 2–1.

[KA17] Sudeep Kanav and Vincent Aravantinos. Modular transformation from AF3 to nuXmv. In: *MODELS (Satellite Events)*, pp. 300–306. 2017.

[Ke+08] X. Ke, P. Pettersson, K. Sierszecki, and C. Angelov. 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. doi: 10.1109/RTCSA.2008.32.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In: G. Gopalakrishnan and S. Qadeer (eds.), *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, LNCS, vol. 6806, pp. 585–591. Springer, 2011.

[Kru+15] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing* 17, 2015. 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian, pp. 184–206. doi: https://doi.org/10.1016/j.pmcj.2014.09.009.

[KSA07] X. Ke, K. Sierszecki, and C. Angelov. 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. doi: 10.1109/RTCSA.2007.29.

[KSH09] Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Tool support for the rapid composition, analysis and implementation of reactive services. *Journal of Systems and Software* 82(12), 2009, pp. 2068–2080. doi: https://doi.org/10.1016/j.jss.2009.06.057.

[KTK09] M. Kadono, T. Tsuchiya, and T. Kikuno. Using the NuSMV model checker for test generation from statecharts. In: *15th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 37–42. 2009.

[Lam09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* 1st. Wiley Publishing, 2009.

[LB13] Bruno Legeard and Arnaud Bouzy. Smartesting CertifyIt: model-based testing for enterprise it. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 391–397. 2013. doi: 10.1109/ICST.2013.55.

[LD01] Wayne Liu and P Dasiewicz. Component interaction testing using model-checking. In: *Canadian Conference on Electrical and Computer Engineering Conference Proceedings (Cat. No. 01TH8555)*, vol. 1, pp. 41–46. 2001.

[Lem+13] Rogério de Lemos et al. Software engineering for self-adaptive systems: a second research roadmap. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers.* Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Springer Berlin Heidelberg, 2013, pp. 1–32. doi: 10.1007/978-3-642-35813-5_1.

[Len20] Dénes Lendvai. *Scenario-Based Modeling and Analysis of Reactive Systems.* Tech. rep. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2020.

[Len22] Dénes Lendvai. *Efficient Scenario-Based Verification of Reactive Systems in the Gamma Framework.* Tech. rep. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2022.

[LLS18] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In: Vladimir Itsykson, Andre Scedrov, and Victor Zakharov (eds.), *Tools and Methods of Program Analysis*, pp. 77–89. Springer International Publishing, 2018. doi: 10.1007/978-3-319-71734-0_7.

[LMM99] Diego Latella, István Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In: *Formal Methods for Open Object-Based Distributed Systems*, pp. 331–347. Springer, 1999.

[LNN] Tim Lange, Martin R Neuhauber, and Thomas Noll. IC3 software model checking on control flow automata. In: *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 97–104.

[LP95] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE* 83(5), 1995, pp. 773–801. doi: 10.1109/5.381846.

[LSD08] Yang Liu, Jun Sun, and Jin Song Dong. An analyzer for extended compositional process algebras. In: *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pp. 919–920. Association for Computing Machinery, 2008. doi: 10.1145/1370175.1370187.

[Lug+16] F. Lugou, L. W. Li, L. Apvrille, and R. Ameur-Boulifa. SysML models and model transformation for security. In: *Conf. on Model-Driven Engineering and Software Development (Modelsward'2016)*, 2016.

[Maz+16] Silvia Mazzini, John M. Favaro, Stefano Puri, and Laura Baracchi. CHESS: an open source methodology and toolset for the development of critical systems. In: *EduSymp/OSS4MDE@MoDELS*, 2016.

[Miy+19] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. Robochart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling* 18(5), 2019, pp. 3097–3149.

[MM19] Vince Molnár and István Majzik. Saturation enhanced with conditional locality: application to Petri Nets. In: *International Conference on Applications and Theory of Petri Nets and Concurrency*, pp. 342–361. 2019. DOI: 10.1007/978-3-030-21571-2_19.

[Moh+14] Swarup Mohalik, Ambar A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh. Automatic test case generation from Simulink/Stateflow models using model checking. *Softw. Test. Verif. Reliab.* 24, 2014, pp. 155–180.

[Mon20] Milán Mondok. *Extended symbolic transition systems: an intermediate language for the formal verification of engineering models.* Tech. rep. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2020. URL: https://tinyurl.com/2p8bvd96.

[Mor+15] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 1–12. Association for Computing Machinery, 2015. DOI: 10.1145/2786805.2786853.

[MPS07] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Behavioral adaptation of component compositions based on process algebra encodings. In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pp. 385–388. Association for Computing Machinery, 2007. DOI: 10.1145/1321631.1321690.

[Nag+21] Simon József Nagy, Richárd Szabó, Máté Levente Vajda, and András Vörös. Demonstrator for dependable edge-based cyber-physical systems. In: *2021 10th Latin-American Symposium on Dependable Computing (LADC)*, pp. 1–8. 2021.

[Nuz+15] Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, Davide Bresolin, Luca Geretti, and Tiziano Villa. A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. *Proceedings of the IEEE* 103(11), 2015, pp. 2104–2132.

[Pan01] Preeti Ranjan Panda. SystemC: a modeling platform supporting multiple design abstractions. In: *Proceedings of the 14th international symposium on Systems synthesis*, pp. 75–80. 2001.

[PF20] Sam Procter and Peter Feiler. The AADL error library: an operationalized taxonomy of system errors. *Ada Lett.* 39(1), 2020, pp. 63–70. DOI: 10.1145/3379106.3379113.

[Pin07] Gergely Pintér. Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems. Ph.D. thesis. Budapest University of Technology and Economics, Hungary, 2007.

[Qur+10] Nauman A. Qureshi, Anna Perini, Neil A. Ernst, and John Mylopoulos. Towards a continuous requirements engineering framework for self-adaptive systems. In: *2010 First International Workshop on Requirements@Run.Time*, pp. 9–16. 2010. DOI: 10.1109/RERUNTIME.2010.5628552.

[Rad22] Bálint Radnai. *Integration of SCXML State Machines to the Gamma Framework*. Tech. rep. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2022. URL: https://tinyurl.com/4mmtsw7v.

[RD12] E. J. Rapos and J. Dingel. 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. DOI: 10.1109/ICST.2012.205.

[Rei17] Wolfgang Reisig. *Associative Composition of Reactive Systems*. Tech. rep. Humboldt-Universität zu Berlin, Institut für Informatik, 2017, pp. 1–18. URL: https://pdfs.semanticscholar.org/224e/8047fd7dd6394e0b0eb15e8328a814c3f5f1.pdf.

[RH] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In: *Proceedings. Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001*, pp. 83–91.

[Rou+09] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Springer Berlin Heidelberg, 2009, pp. 164–182. DOI: 10.1007/978-3-642-02161-9_9.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development* 3(2), 1959, pp. 114–125. DOI: 10.1147/rd.32.0114.

[RW85] Sandra Rapps and Elaine Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* SE-11(4), 1985, pp. 367–375. DOI: 10.1109/TSE.1985.232226.

[Sal+17] Yasir Dawood Salman, Nor Laily Hashim, Mawarny Md Rejab, Rohaida Romli, and Haslina Mohd. Coverage criteria for test case generation using UML state chart diagram. *AIP Conference Proceedings* 1891(1), 2017, p. 020125. DOI: 10.1063/1.5005458.

[Sel98] Bran Selic. Using UML for modeling complex real-time systems. In: Frank Mueller and Azer Bestavros (eds.), *Languages, Compilers, and Tools for Embedded Systems*, pp. 250–260. Springer Berlin Heidelberg, 1998.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-oriented Modeling*. John Wiley & Sons, Inc., 1994.

[Sim+13] Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, and Janos Sztipanovits. Specification of Cyber-Physical Components with Formal Semantics – Integration and Composition. In: Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke (eds.), *Model-Driven Engineering Languages and Systems*, pp. 471–487. Springer, 2013.

[SK06] Elisabeth A. Strunk and John C. Knight. Dependability through assured reconfiguration in embedded system software. *IEEE Trans. Dependable Secur. Comput.* 3(3), 2006, pp. 172–187. DOI: 10.1109/TDSC.2006.33.

[SL15] Muhammad Shafique and Yvan Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer* 17(1), 2015, pp. 59–76. DOI: 10.1007/s10009-013-0291-0.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press, 1988.

[SP10] Francesca Saglietti and Florin Pinte. Automated unit and integration testing for component-based software systems. In: *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*, pp. 1–6. 2010.

[SST06] Klaus Schneider, Tobias Schuele, and Mario Trapp. Verifying the adaptation behavior of embedded systems. In: *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems*, SEAMS '06, pp. 16–22. Association for Computing Machinery, 2006. DOI: 10.1145/1137677.1137681.

[Ste+08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[SW07] Wilhelm Schafer and Heike Wehrheim. The challenges of building advanced mechatronic systems. In: *Future of Software Engineering (FOSE '07)*, pp. 72–84. 2007. DOI: 10.1109/FOSE.2007.28.

[Szk22] Péter Szkupien. *Step-by-step controllable simulation of component-based reactive systems based on precise formal semantics*. Tech. rep. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2022.

[Szt+14] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems. In: *From Programs to Systems. The Systems perspective in Computing. ETAPS Workshop, FPS 2014*. Ed. by Saddek Bensalem, Yassine Lakhneck, and Axel Legay. Springer, 2014, pp. 235–248.

[Tam+13] Gabriel Tamura et al. Towards practical runtime verification and validation of self-adaptive software systems. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Springer Berlin Heidelberg, 2013, pp. 108–132. DOI: 10.1007/978-3-642-35813-5_5.

[Tót+17] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In: Daryl Stewart and Georg Weissenbacher (eds.), *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pp. 176–179. 2017. DOI: 10.23919/FMCAD.2017.8102257.

[Tra+07] Mario Trapp, Rasmus Adler, Marc Förster, and Janosch Junger. Runtime adaptation in safety-critical automotive systems. In: *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*, SE'07, pp. 308–315. ACTA Press, 2007. DOI: 10.5555/1332044.1332094.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Elsevier, 2007. DOI: 10.1016/B978-0-12-372501-1.X5000-5.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability* 22(5), 2012, pp. 297–312. DOI: 10.1002/stvr.456.

[Var+16] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling* 15(3), 2016, pp. 609–629.

[VG14] Thomas Vogel and Holger Giese. Model-driven engineering of self-adaptive software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* 8(4), 2014. doi: 10.1145/2555612.

[Vul+09] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the ProCom real-time component model. In: *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 478–485. 2009.

[Wag92] F. Wagner. VFSM executable specification. In: *CompEuro 1992 Proceedings Computer Systems and Software Engineering*, pp. 226–231. 1992.

[Wey+12] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. A survey of formal methods in self-adaptive systems. In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pp. 67–79. Association for Computing Machinery, 2012. doi: 10.1145/2347583.2347592.

[Wey20] Danny Weyns. *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective.* John Wiley & Sons, 2020.

[Wie+09] Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In: Manuel Núñez, Paul Baker, and Mercedes G. Merayo (eds.), *Testing of Software and Communication Systems*, pp. 179–194. Springer Berlin Heidelberg, 2009.

[Zav21] Ármin Zavada. *Formal Modeling and Verification of Process Models in Component-based Reactive Systems.* Tech. rep. Budapest Univ. of Technology, Economics, Dept. of Measurement, and Information Systems, 2021.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In: *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 371–380. Association for Computing Machinery, 2006. doi: 10.1145/1134285.1134337.

[ZD17] Karolina Zurowska and Juergen Dingel. Language-specific model checking of UML-RT models. *Software & Systems Modeling* 16(2), 2017, pp. 393–415.

[Zur14] Karolina Zurowska. Language Specific Analysis of State Machine Models of Reactive Systems. Ph.D. thesis. Queen's Univerity, Canada, 2014.