Integration test generation for state-based components in the Gamma framework

Bence Graics[1*], Vince Molnár[1†] and István Majzik[1†]

[1]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Magyar Tudósok körútja 2. Building I, Wing E, Floor IV, Room IE444, Budapest, H-1117, Hungary.

*Corresponding author(s). E-mail(s): graics@mit.bme.hu;
Contributing authors: molnarv@mit.bme.hu; majzik@mit.bme.hu;
†These authors contributed equally to this work.

Abstract

Nowadays, the design of complex reactive systems, e.g., control systems in the railway and aerospace industries, is generally based on the integration of components. Components (subsystems) may come from various parties and operate according to different execution and interaction semantics. Analyzing the joint behavior of such integrated components is a tedious task. This paper introduces a model-based approach in our Gamma Statechart Composition Framework for integration test generation on the basis of collaborating state-based models. Test generation is supported by test coverage criteria and the automatic transformation of composite models into different model checker backends. The diagnostic traces returned by the model checkers are automatically mapped into test cases. As novelty, our tool is based on *precise composition semantics* (synchronous and asynchronous) and relies on *model queries* to support the *modular* and *flexible* specification of element coverage according to various model-based criteria: dataflow, component interactions and static model elements. The tool utilizes *model reduction* and *slicing* techniques during the transformations to ease test generation complexity using query-dependent model processing. We demonstrate the applicability of our tool on three industrial subsystems, including an aerospace (NASA) and two railway applications.

Keywords: model-based integration testing, modular coverage criteria, model checkers, integrated MBT tool

1 Introduction

Model-based systems engineering (MBSE) and component-based systems engineering (CBSE) [1–6] are getting more prevalent in the design of complex reactive systems, e.g., critical control systems in the railway and aerospace industries. The design of such systems is increasingly based on the integration of components defined in high-level modeling languages preferably with automatically generated implementation. The components may come from various sources, creating a loosely coupled distributed architecture.

In different subsystems, various execution and interaction semantics may be used, e.g., asynchronous communication based on immutable messages stored in message queues or synchronous communication using sampled signals. Such characteristics encumber the integration process as precise hierarchical integration according to different composition semantics is poorly supported by

present-day tools [7]. The lack of formally defined execution and interaction modes also hinders the precise verification of the resulting system. Thus, system integrators generally have to rely on their intuition and test the integrated system based on typical system use cases (commonly provided by domain experts), which may be informal, incomplete and obtained in an ad hoc way.

Systematic analysis could be supported by employing model-based testing (MBT) methods [8], which can generate tests relying on different techniques and systematically provide the coverage of structural or behavioral criteria in the model, e.g., state, transition or interaction coverage [9]. In addition, formal methods (e.g., model checking) based solutions can also prove the unreachability of test targets. However, it is hard to find an MBT tool that supports integration test generation according to various and mixed composition semantics [7]. Furthermore, the configuration and the introduction of new test criteria are generally cumbersome or not supported.

Thus, MBT approaches and tools aiming to aid the systematic integration testing of reactive components should support *(i)* the *well-defined integration of components* (potentially created in different tools) according to different *execution and interaction semantics*, *(ii)* precisely defined, various *test criteria* for structural model elements, component behavior (interactions) and dataflow, and *(iii) efficient test generation* algorithms in terms of execution time and the size of generated test sets (test optimization). As a solution, we propose an approach in the context of the open-source Gamma Statechart Composition Framework.

Our Gamma Statechart Composition Framework[1] [10] is an integrated modeling toolset supporting the semantically well-founded composition and analysis of heterogeneous statechart components. At its core, it provides a composition language supporting the hierarchical mixed-semantics interconnection of components [11]. Gamma supports system-level formal verification and validation (V&V) by mapping statechart and composition models into analysis formalisms of various model checkers and back-annotating the results to an abstract trace language.

[1] Additional information about the framework and the source code can be found at http://gamma.inf.mit.bme.hu/ and https://github.com/ftsrg/gamma/.

This paper presents a self-contained automatic MBT solution in the Gamma framework by extending its general component integration and verification functionalities [10, 11]. By building on Gamma, our test generation approach utilizes the formally defined and hierarchical mixed-semantics composition of heterogeneous state-based components (potentially with time-dependent behavior). It also utilizes the integrated model checker backends of Gamma to generate diagnostic traces that are automatically mapped into test cases.

In the following, we will use the term *"Gamma test generator tool"* (or *"tool"* in short) to refer to the new *test generator application* presented in this paper and use the term *"Gamma framework"* or simply *"Gamma"* or *"framework"* when we refer to all the functionalities that Gamma as an integrated modeling toolset can offer (and the test generator tool builds on).

As novelty, in our tool we introduced the following features to aid the model-based systematic integration testing of reactive components.

- By building on and extending coverage criteria presented in the literature, for integration testing we introduced, formally defined and implemented dataflow-based, structural (model element-based) and behavior-based (interactional) *coverage criteria* in the context of *interconnected state-based* models. Based on these criteria, the generated tests can detect faults in component implementations (e.g., missing implementation of transitions), interaction of components, and improper variable definitions and uses in composite system implementations.
- The tool supports the *configuration* of coverage criteria: coverable elements can be specified and adjusted by users in flexible ways, facilitating testing focus on specific system parts (components). As a key aspect, the tool utilizes *model queries* for specifying coverage criteria and their automatic mapping into formal (model checking) properties. Model queries are extensible, allowing the easy customization of available and the introduction of new criteria. As queries are defined in the form of graph patterns, they facilitate the high-level, declarative specification and automatic exploration of non-trivial interrelations between model elements.
- To improve the efficiency of test generation, the tool applies generic and criterion-dependent

model reduction and *slicing* techniques, relying on model queries in the exploration of removable model elements.
- The tool integrates multiple *model checker backends* (selectable target formal models) that complement each other in supporting various model types and composition semantics. We provide *practical experiences* regarding time-efficiency of test generation on real-world problems, and evaluate the advantages of different supported formal target models (providing hints for selection) and model processing methods.

With respect to the state-of-the-art, these contributions can be considered partly incremental (like model queries, reductions and slicing), and partly based on well-known methods (like test case generation using model checkers). However, to the best of our knowledge, there is no open-source tool that is based on mixed formal composition semantics, supports configurable interaction-oriented test criteria, and implements efficiency-improving techniques seamlessly in a single modeling framework – together with source code generation and formal verification that are also supported by Gamma.

In terms of MBT, the initial Gamma versions presented in [10] and [11] supported only *unconfigurable* simple structural (state- and transition-based) test criteria without providing (the more complex and cumbersome) interaction and inter-component dataflow coverage. The efficient implementation of test generation based on these new criteria required the new model query based approach as well as model reduction and slicing.

Although we present these solutions in our Gamma tool, the contributions regarding the test coverage criteria, model queries and model processing techniques are general (build on graph languages) and could be applied in other MBT approaches and integrated into other tools, too.

The rest of the paper is structured as follows. Section 2 overviews component integration and verification in the Gamma framework, which serve as a basis for our test generation approach presented in Sect. 3. Section 4 formally defines the supported test coverage criteria and overviews how composite models are annotated and properties are generated to specify coverable test targets. Section 5 presents the processing of annotated composite models, properties and resulting abstract test cases during test generation. Section 6 introduces the versatility and efficiency of our tool by generating tests according to different coverage criteria for industrial subsystems, including a NASA-related aerospace application and two railway-related subsystems. Section 7 covers related work. Finally, Sect. 8 concludes the paper and presents possible directions for future work.

2 Component integration and verification in Gamma

This section overviews the general workflow according to which components can be integrated and verified in Gamma,[2] serving as a basis for our test generation approach detailed in later sections. We introduce the workflow in the context of a simplified *elevator system* first presented in [9]. The components of the system are modeled in Yakindu (see Fig. 1) comprising a *cabin controller* that 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.

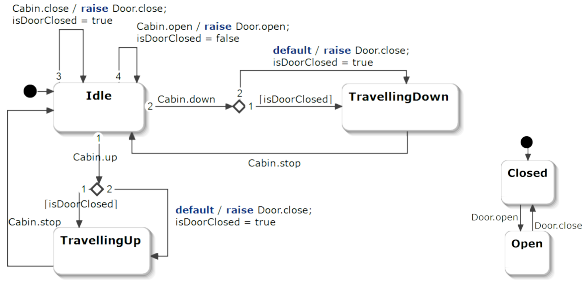

Fig. 1 *Cabin controller* and *cabin door controller* models of the *elevator system*.

The workflow builds on a model transformation chain depicted in Fig. 2, which illustrates the input and output models of these model transformations as well as the languages in which they are defined, and the relations between them. The modeling languages are as follows.

[2]Even though, the Gamma framework currently supports statechart models as input, its modeling language family supports the integration of other formalisms, e.g., we are working on integrating activity diagrams [12].

- The **Gamma Statechart Language (GSL)** is a UML/SysML-based statechart language supporting different semantic variants of statecharts.
- 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.
- The **Gamma Genmodel Language (GGL)** is a configuration language for configuring model transformations.
- The **Gamma Property Language (GPL)** is a property language supporting the definition of CTL* [13] properties and thus, the formal specification of requirements regarding (composite) component behavior.
- The **Gamma Trace Language (GTL)** is a high-level specification language for execution traces of (composite) components.

The following sections present the consecutive steps (identified with numbers in the overview figure) of the general component integration and verification workflow in the Gamma framework. Note that the sections include only high-level descriptions of the functionalities; for more detailed descriptions, we direct the reader to the Appendix and [11].

Optionally, statechart models defined in supported modeling tools (front-ends) can be imported into Gamma (Sect. 2.1), which can be integrated according to well-defined execution and interaction semantics (Sect. 2.2). The resulting composite model is processed and transformed into the input formalisms of integrated model checker back-ends (Sect. 2.3). The model checker back-ends provide witnesses (diagnostic traces) based on specified properties, which are back-annotated, resulting in abstract traces (Sect. 2.4) In transition to the subsequent sections, Sect. 3 introduces the general ideas on how our test generation approach extends the general integration and verification workflow to derive coverage-based test sets for composite model implementations.

2.1 Importing external component models (optional)

The import of an external component model, i.e., a statechart created in an external modeling tool, is realized by executing a model transformation that maps this model into a GSL statechart. Currently, the import of Yakindu[3] and MagicDraw[4] models are supported, but the integration of other UML/SysML tools would be only a technological challenge. GSL serves as a common representation language for component statecharts and supports different statechart semantics by means of *annotations*: *conflict resolution* between transitions of different hierarchy levels (*top-down* or *bottom-up* execution) and the *definition of potential priorities* between transitions with the same source (nondeterministic choice, priority in the order of definition or priority based on explicit integer values). 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 Figs. 16 and 17 in the Appendix, respectively.

2.2 Integrating component models

Gamma statecharts, which may be derived from external statechart models (see Step 1) or defined directly in GSL, can be hierarchically integrated according to various precise execution and interaction semantics in GCL to create *synchronous* or *asynchronous* systems; the resulting composite models are validated against well-formedness constraints. In synchronous systems, communication is based on signal transmission and sampling, and can be modeled using the *synchronous-reactive* and *cascade* composition modes. In asynchronous systems, communication is based on the delivery (dispatch and reception) of immutable messages, supported by the *asynchronous-reactive* composition mode. The introduction and formal semantics of these composition modes can be found in [11]; here we include a summary of their properties.

- *Synchronous-reactive* composite models represent a coherent unit consisting of strongly coupled components, which are executed *concurrently* in a lockstep fashion and communicate in a synchronous manner using signals.
- The *cascade* composition mode is a variant of the synchronous-reactive mode: components in a cascade model are executed in a *sequential*

[3] https://www.itemis.com/en/yakindu/state-machine/
[4] https://www.nomagic.com/products/magicdraw/

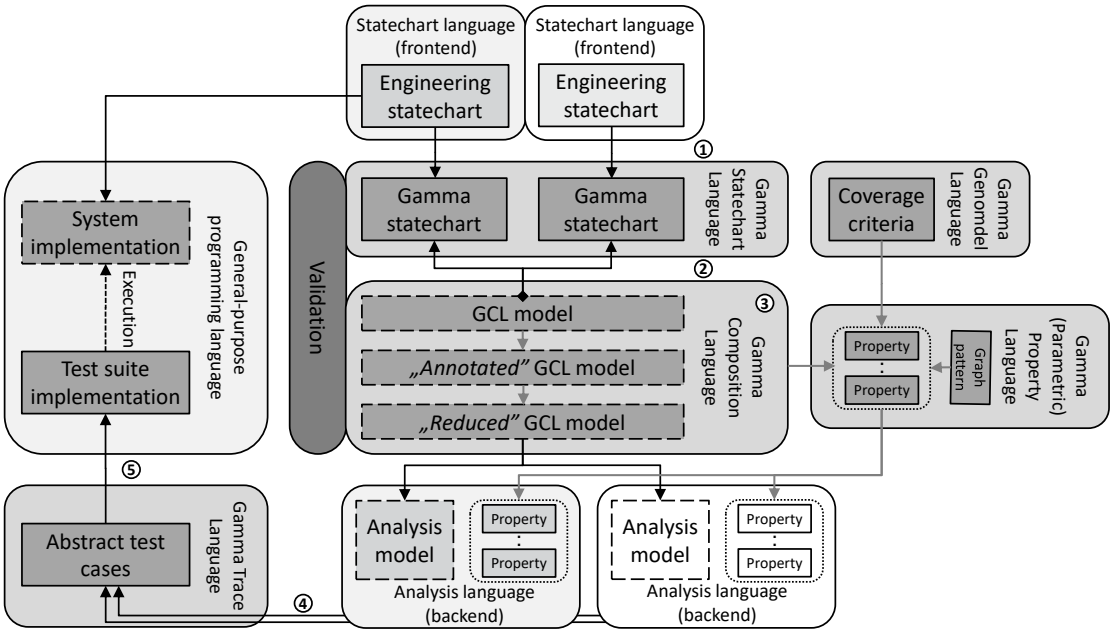

Fig. 2 Model transformation chains and modeling languages of the test generation approach in the Gamma framework. Rectangles represent models: solid borders represent atomic models, whereas dashed borders represent composite models. Dotted rectangles represent a set of models belonging together for fulfilling a more general purpose. Rectangles with moderately rounded corners represent modeling languages. Rectangles with extensively rounded corners represent functionalities closely related to the usability of a language. Solid lines without a base symbol represent model transformations. Solid lines with a diamond symbol represent model composition. Dashed lines represent the ability of execution: the source artifact is capable of executing the target artifact. Black color in lines represents relations to the general integration and verification workflow whereas grey represents relations to the test generation approach (extensions to the original workflow).

manner. Contained components can be considered as a set of filters applied sequentially to derive an output from an input.

- *Asynchronous-reactive* composite models represent a collection of independently (parallelly) running components. There is no guarantee on the execution time or the execution frequency of such components, thus, they communicate with queued (persistent) messages.

Synchronous-reactive and *cascade* composition modes provide deterministic behavior and thus, such models are a suitable target for formal verification. However, *asynchronous-reactive* models are inherently nondeterministic with potential interleavings and thus can generally pose a great challenge for model checkers [14, 15]. Nevertheless, Gamma supports them as distributed systems frequently follow such semantics and the modeling and code generation functionalities of the framework can be of use to practitioners even when formal verification is not expected.

The GSL models of our elevator example generated in Step 1 can be integrated in GCL to create *synchronous-reactive*, *cascade* or *asynchronous-reactive* composite models using a textual syntax presented in Fig. 18 (Appendix).

2.3 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 as proof [16]. Gamma facilitates deriving both the analysis model and the property using automated model transformations.

To derive analysis models, the composite models created in Step 2 are preprocessed and transformed into the input languages of integrated model checker back-ends. The transformation can be configured in GGL in a textual format.

For example, the following snippet specifies that the *Elevator* composite model is to be transformed into the input formalism of UPPAAL.

```
component : Elevator
language : UPPAAL
```

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* 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.

```
@("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. To reduce the state space of the model, the transformations exploit optimization techniques based on variables that store *resettable* or *transient* data.

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 turn (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 preprocessing steps, e.g., for boolean variables created for transitions annotated with an identifier.

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.

2.4 Executing model checking 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[5] [17] and Theta[6] [18] are integrated as back-ends, but the possibility of integrating additional model checkers is allowed by the framework.

Gamma supports the tuning of the model checking process, e.g., the specification of search strategies, such as breadth-first search or depth-first search, 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 in a textual format[7] based on the traceability links defined in the corresponding composite model transformation.

The snippet in Fig. 19 in the Appendix describes GTL execution traces derived on the basis of *E F (var main_open__main_closed)* and *E F (var main_closed__main_open)* GPL properties.

3 Test generation in Gamma

Our test generation approach offers *structural*, *dataflow* and *behavior-related* coverage based automatic test generation for interconnected state-based components, focusing on interactions specified by the system model. The generated tests can detect faults in automatically generated or manually created component implementations (e.g., missing state and transition implementations as well as improper variable definitions and

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

[6] http://theta.inf.mit.bme.hu/

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

uses), composite system implementations (e.g., interactions between components) and in the integration of the execution platform (e.g., communication libraries, middlewares).

As a general idea, in a testing context, an execution trace for a composite model derived during formal verification can be considered as an *abstract test case* for the property based on which it is generated, representing a test target. Thus, with the goal of generating tests while building on the functionalities of the Gamma framework, we can control model checkers in a way that they generate execution traces (abstract test cases) to cover test targets specified as formal properties [19]. In this context, we use Gamma's trace language as an abstract test language and also refer to it as **Gamma Test Language (GTL)**.

Such an abstract test case defined in GTL can be customized to concrete test environments according to various aspects [20]. For example, if one wants to consider only outputs in response to incoming inputs during testing (e.g., because the implementation realizes a different internal structure) then internal state assertions can be easily discarded from the abstract test cases. Gamma currently supports Java as an execution environment and JUnit[8] as a test environment, but additional environments could also be integrated. The framework provides a flexible reflective Java API for state-based implementations designed for simulation. 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. As an example, the snippet in Fig. 20 in the Appendix describes the JUnit class derived from *ClosedOpenTrace* defined in Fig. 19. Further information on GTL and the Java API can be found in [21].

Our test generation approach extends Gamma's general component integration and verification workflow to support a self-contained *test generation workflow* for state-based systems (see Fig. 2) using the precise semantics composite models as reference specifications. The steps of the test generation workflow are as follows.

1. The composite model is manually designed in GCL by (hierarchically) integrating GSL statechart components (manually created, or imported from the integrated modeling front-ends – Step 1 in Fig. 2) according to the various supported composition semantics (Step 2). The models are validated both at statechart and composite component level.
2. Optionally, based on the resulting composite model, formal verification using model checkers (Steps 3 and 4) and code generation can be carried out in an automated way.
3. As the first test generation specific step, test targets for the composite model are specified, including the manual specification and configuration of selectable test coverage criteria (e.g., filtering model elements for testing).
4. As an automated series of steps, the tool
   (a) maps these high-level test targets to formal properties,
   (b) processes the input composite model in accordance with the selected test targets (model annotation) and reduction techniques, derives analysis models, and
   (c) uses the integrated model checker back-ends in an optimized way to generate execution paths in the generated analysis models for the coverage of these test targets (see grey elements in Fig. 2), deriving abstract tests.
5. The derived abstract tests then (after potential optimizations) are automatically concretized to execution environments (Step 5).

The test generation features (detailed in subsequent sections) supporting Steps 3 and 4 of the test generation workflow incorporate

- configurable *coverage criteria* and automated *model annotation* techniques to enable the explicit formulation of test targets in composite models and the mapping of these targets into formal *reachability properties* (detailed in Sect. 4), and
- *model reduction* and *model slicing* algorithms based on model queries to reduce the composite model and assist model checking in addition to *optimization techniques* in the abstract test generation process (detailed in Sect. 5) to decrease both the time of the generation process and the size of the generated test sets.

[8] https://junit.org/

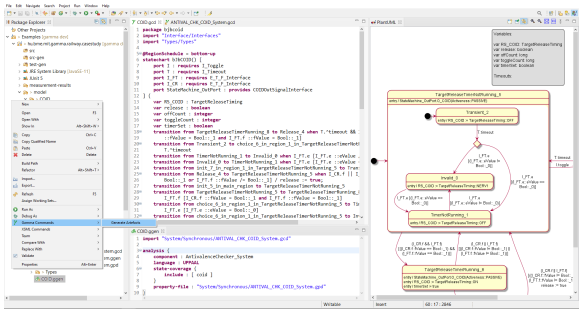

Fig. 3 Graphical user interface (GUI) of Gamma.

This way, all Gamma functionalities inherently become available for test generation in the workflow, i.e., the import of external models, their mix-and-match composition according to precise composition semantics in addition to the derivation of concrete test cases potentially to multiple environments. Conversely, the generated tests can also be used to validate the sequence of internal model transformations as well as code generators of Gamma by demonstrating semantic equivalence between model and code for the examined traces.

Figure 3 depicts how the Gamma GUI (integrated into Eclipse) can be used to generate state-covering test sets (see *COID.ggen* file in the text editor) targeting a statechart component (see *COID.gcd*) in a composite model.

4 Coverage criteria and their mapping into test targets

This section presents the *coverage criteria* of our test generation approach and the related *model annotation* and *property generation* techniques facilitating the explicit formulation of test targets. Section 4.1 formalizes the built-in test coverage criteria in the context of composite models. Section 4.2 overviews the semantic-independent injection of auxiliary fragments (annotations) into composite models according to the selected coverage criteria, enabling the explicit formulation of test targets. Section 4.3 presents the generation of formal reachability (temporal logic) properties, specifying the coverage of the annotated elements.

4.1 Specification of coverage criteria

With our approach, we aim to facilitate flexible and modular test generation according to multiple aspects relevant in the context of reactive systems and support the following coverage criteria that are grouped into three different classes:

- **structural** coverage criteria include *state*, *transition*, *transition-pair* (incoming and outgoing transition pairs for states) [22] and *out-event* coverage,
- **behavior-based** coverage criteria include *interaction* coverage [9], that is, the sending and the reception/processing of an event between two communicating components according to configurable combinations, and
- **dataflow-based** coverage criteria [23] include the coverage of execution paths between the definition (*def*) and the use/reading (*use*) of variables within a single component and also between communicating components.

Test coverage criteria for composite models can be specified in two ways: *(i) predefined* criteria can be configured using the *GGL-based approach*, or *(ii) custom* structural criteria can be specified based on model queries using the parametric property language of Gamma (*GPPL-based approach*).

GGL-based criterion specification

In the GGL-based approach, the coverage criteria are specified in a configuration file. In this file (see Fig. 4), the composite model is specified first, which is followed by a keyword denoting the selected predefined coverage criterion. In order to reduce the number of unnecessary tests, the criteria can be configured by specifying components (both composite and atomic components are supported) that must be included or excluded from the composite model (*include* and *exclude* keywords). The approach supports further adjustments based on relevant model elements of certain criteria. Note that all these configuration possibilities are defined precisely in Sect. 4.1.2. In every case where model elements are included or excluded, inclusion has a higher priority than exclusion. In case no inclusion or exclusion is explicitly specified, every element (component, port, state, transition or variable) in the composite model is considered for test generation.

GPPL-based criterion specification

To support the custom specification of test coverage criteria in a declarative way, we introduce the **Gamma Parametric Property Language**

```
component : gclModel
[state / transition / transition-pair / out-event /
 interaction / dataflow / interaction-dataflow]-coverage {
 include : [includedComponent, otherIncludedComponent]
 exclude : [excludedComponent]
 // Only for transition, transition-pair and interaction
      coverage
 transition-include : [atomic.aTransition]
 transition-exclude : [atomic.anotherTransition]
 // Only for out-event, interaction and interaction-
      dataflow coverage
 port-include : [atomic.includedPort]
 port-exclude : [atomic.excludedPort]
 // Only for interaction coverage
 sender-coverage-criterion : [events /
   states-and-events / every-interaction]
 receiver-coverage-criterion : [events /
   states-and-events / every-interaction]
 state-include : [atomic.aState]
 state-exclude : [atomic.anotherState]
 // Only for dataflow and interaction-dataflow coverage
 coverage-criterion: [all-def / all-c-use /
   all-p-use / all-use]
 // Only for dataflow coverage
 variable-include : [atomic.aVariable]
 variable-exclude : [atomic.anotherVariable]
}
```

Fig. 4 Textual syntax to specify coverage criteria in GGL.

(GPPL) that extends GPL with *model queries*, supporting the specification of model element sets with arbitrary interconnections (i.e., coverable elements) *independently* of concrete composite models. Based on such queries, the language supports the definition of *parameterized* GPL expressions where the parameters refer to model elements (e.g., states or transitions) specified by model queries. This way, as the queries in the case of concrete composite models provide sets of model elements, the parameterized GPL expressions are instantiated, deriving *concrete* GPL properties.

GPPL currently expects model queries to be defined as graph patterns in a custom query language (see Fig. 5) and supports the specification of interconnections between states and transitions. Nevertheless, experienced users, who have willingness to use the open internal APIs of Gamma, may also apply the generic query language of VIATRA (see Sect. 4.2.2 for details), this way utilizing greater flexibility and expressive power for query definitions.

In the query language, a pattern definition has an identifier (name), followed by parameter declarations of type *State* or *Transition* (elements targeted for coverage). Interconnections between the parameters can be specified in the pattern body (constraints are in an And-relation). Currently, for state parameters, the language supports

```
pattern errorStates(s : State) {
 s.name matches ".*Error" // Regex for name
}
pattern transitionsToLoopTransition(
    t1 : Transition, t2 : Transition) {
 t1.source.name matches ".*Error" // Regex
 t1.source != t1.target // Non-loop
 t1.target = t2.source  // Connecting t1-t2
 t2.source = t2.target // Loop transition
}
E F s // Covering error states
E F t1 // Covering transitions leaving an error state and
     entering a state with a loop transition
```

Fig. 5 Syntax for specifying coverage criteria in GPPL.

referencing their *name* attribute (of type string). For transition parameters, *source* and *target* references are supported, which behave as state references, i.e., their name attribute can be referenced. Such references can be compared using equality and inequality operators, and strings can be matched against regular expressions (*matches* keyword). These constructs enable the specification of complex criteria describing, e.g., the coverage of states with specific names and loop or non-loop transitions.

Building on these patterns, *parameterized* GPL expressions can be defined in which pattern parameters of states and transitions may be used instead of concrete state and transition references. During test generation, unique concrete GPL expressions (duplications are filtered) are generated from these expressions by replacing the state and transition parameters with corresponding elements from the bound composite model, serving as test targets.

Compared to the GGL-based approach, GPPL specifications are independent of any (testable) composite models and support a more flexible way to specify coverage criteria. The language also allows for potential modularization methods and easy addition of future extensions.

4.1.1 Formal concepts of coverage criteria

In order to facilitate the formal specification of the proposed coverage criteria, we introduce the following concepts related to the coverage of model elements in composite models.

An *atomic (statechart) component* in a composite model with unique index i is denoted $A_i = (S_i, I_i, O_i, T_i)$ where

- $S_i = \{s_{i,0}, \cdots, s_{i,m}\}$ denotes the finite set of states of the atomic component with initial state $s_{i,0}$.
- $P_i = \{p_{i,1}, \cdots, p_{i,q}\}$ denotes the finite set of pseudostates (entry, choice and merge states) of the atomic component.
- $N_i = S_i \cup P_i$ denotes the finite set of nodes (states and pseudostates) of the atomic component.
- $I_i = \{i_{i,1}, \cdots, i_{i,k}\}$ denotes the finite set of input events of the atomic component.
- $O_i = \{o_{i,1}, \cdots, o_{i,l}\}$ denotes the finite set of output events of the atomic component.
- $T_i = \{t_{i,1}, \cdots, t_{i,n}\}$ denotes the finite set of transitions of the atomic component where $t_{i,j} = (l_{i,j}, e_{i,j}) \in N_i \times N_i$, that is, a transition has a source state $l_{i,j} \in N_i$ and a target state $e_{i,j} \in N_i$.
- $E_i : T_i \to 2^{I_i}$ denotes the input events (independently) triggering a certain transition.
- $R_i : (T_i \cup S) \to 2^{O_i}$ denotes the output events raised by a certain transition or state (entry and exit actions).

A *composite model* as the composition of atomic components $A_1, \cdots, A_n$ is denoted $M = (S, T)$ where

- $S \subseteq S_1 \times \cdots \times S_n$ denotes the finite set of potential composite states of the composite model where the initial state is $s_0 = (s_{1,0}, \cdots, s_{n,0})$.
- $T \subseteq \tau_1 \times \cdots \times \tau_n$ denotes the finite set of potential composite transitions of the composite model where either $\tau_{i,j} = t_{i,j} = (l_{i,j}, e_{i,j}) : l_{i,j}, e_{i,j} \in S_i$ (atomic transition) or $\tau_{i,j} = (t_{i,k}, p_{i,k}, \cdots, p_{i,l-1}, t_{i,l}, p_{i,l}, \cdots, p_{i,m-1}, t_{i,m}) : l_{i,k}, e_{i,m} \in S_i, \forall t_{i,l}, k < l \leq m : e_{i,l} = p_{i,l}, l_{i,l} = p_{i,l-1}$ (a series of atomic transitions connected via pseudostates). From an abstract point of view, the series of atomic transitions $\tau_{i,j}$ is identified by its endpoints $(l_{i,k}, e_{i,m})$.

An *interaction point* between two model elements (sender and receiver) of A_i and A_j where the output of A_i is connected to the input of A_j (with a channel) is denoted $(d_{i,k}, t_{j,l})$ where $d_{i,k} \in (S_i \cup T_i), t_{j,l} \in T_j$ if $E_j(t_{j,l}) \subseteq R_i(d_{i,k})$.

A *valid execution trace* of a composite model M is a finite alternating sequence of composite states and composite transitions starting from the initial state denoted $X = (s_0, t_1, s_1, \cdots, t_j, s_j, \cdots, t_m, s_m)$ where $\forall 0 < j \leq m : s_j \in S, t_j \in T$ and $\forall \tau_{i,l} = (l_{i,l}, e_{i,l}) \in t_j : l_{i,l} \in s_{j-1}, e_{i,l} \in s_j$ (an atomic or composite transition of a component leaves a stable state and enters a stable state).

- An execution trace covers state $s_{i,k} \in S_i$ of atomic component A_i if $\exists j : s_{i,k} \in s_j$.
- An execution trace covers transition $t_{i,k} \in T_i$ of atomic component A_i if $\exists j : t_{i,k} \in t_j$ (the transition is atomic or an element of a series of transitions connected via pseudostates).
- An execution trace covers input event $i_{i,k} \in I_i$ of atomic component A_i if it covers at least one $t_{i,j} \in T_i$ transition where $i_{i,k} \in E_i(t_{i,j})$.
- An execution trace covers output event $o_{i,k} \in O_i$ of atomic component A_i if it covers at least one $t_{i,j} \in T_i$ transition or $s_{i,j} \in S_i$ state where $o_{i,k} \in R_i(t_{i,j})$ or $o_{i,k} \in R_i(s_{i,j})$.
- An execution trace covers interaction point $(d_{i,k}, t_{j,l})$ between A_i and A_j via *interaction* $(o_{i,k}, i_{j,l}) : o_{i,k} \in R_i(d_{i,k}), i_{j,l} \in E_j(t_{j,l})$ if it covers transition $d_{i,k} \in T_i$ or state $d_{i,k} \in S_i$ as well as transition $t_{j,l} \in T_j$ such that $d_{i,k} \in t_p$ or $d_{i,k} \in s_p$ and $t_{j,l} \in t_q, p \leq q$.

4.1.2 Formal definition of coverage criteria

The formal definitions of the proposed coverage criteria based on the introduced concepts considering *execution traces* as *test traces* are the following. Although these definitions build on concepts already presented in the literature, e.g., [9, 22, 23], the formalization of these coverage criteria in the context of *collaborating* (interconnected) state-based components is a novel contribution that precisely defines the (non-trivial) test targets supported by Gamma, but could also be reused in other integration test generation approaches.

State coverage

In the case of *state coverage*, for every atomic component A_i, every state $s_{i,j} \in S_i$ is covered by at least one valid test trace.

Transition coverage

In the case of *transition coverage*, for every atomic component A_i, every transition $t_{i,j} \in T_i$ is covered by at least one valid test trace. In this case, transitions considered or ignored can also be

included or excluded using the *transition-include* and *transition-exclude* keywords.

Transition-pair coverage

In the case of *transition-pair coverage*, for every atomic component A_i, every transition pair $t_{i,j} = (l_{i,j}, e_{i,j}), t_{i,k} = (l_{i,k}, e_{i,k}) \in T_i$ where $e_{i,j} = l_{i,k} : e_{i,j}, l_{i,k} \in S_i$, i.e., incoming and outgoing transition pairs of every potential state, is covered by at least one valid test trace. Similarly to *transition coverage*, individual transitions can be included and excluded using the *transition-include* and *transition-exclude* keywords.

Out-event coverage

In the case of *out-event* coverage criteria, for every atomic component A_i, every output event $o_{i,j} \in O_i$ is covered by at least one valid test trace. In this case, individual ports can be included or excluded via which event raisings are considered or ignored (*port-include* and *port-exclude* keywords).

Interaction coverage

In the case of *interaction coverage*, interaction points are covered according to adjustable options. The options for senders and receivers (*sender-* and *receiver-coverage-criterion*) can be specified independently and in an arbitrary combination. The options are the following (from a more coarse-grained to a finer representation of interactions):

- *events* – every port-event combination in a component is covered (sending or reception according to the role of the component in the interaction),
- *states-and-events* – every port-event combination in every state is covered,
- *every-interaction* – every interaction in a component is covered.

The formalized coverage criteria regarding interaction points from the sender's point of view are as follows. For every A_i and A_j atomic component where output $o_{i,k} \in O_i$ is in interaction with (via a channel) input $i_{j,l} \in I_j$,

- *events* – every $o_{i,k} \in O_i$ is covered by at least one valid test trace via at least one $(o_{i,k}, i_{j,l})$ interaction;
- *states-and-events* – every $(d_i, t_{j,l})$ interaction point is covered by at least one valid test trace where $d_i \in (S_i \cup \dot{\bigcup} T_i)$ such that $t_{i,o} \in q_i$ and $t_{i,p} \in q_i$ where $q_i \in \dot{\bigcup} T_i$ iff $l_{i,o} = l_{i,p} : l_{i,o}, l_{i,p} \in S_i$, and $E_i(t_{i,o}) = E_i(t_{i,p})$ (transitions with the same state source and triggers are not distinguished);
- *every-interaction* – every $(d_{i,k}, t_{j,l})$ interaction point coverable via $(o_{i,k}, i_{j,l})$ is covered by at least one valid test trace.

The coverage criteria regarding interaction points from the receiver's point of view are very similar to that of the sender. For every A_i and A_j atomic component where output $o_{i,k} \in O_i$ is in interaction with input $i_{j,l} \in I_j$,

- *events* – every $i_{j,l} \in I_j$ is covered by at least one valid test trace via at least one $(o_{i,k}, i_{j,l})$ interaction;
- *states-and-events* – every $(d_{i,k}, r_j)$ interaction point is covered by at least one valid test trace where $r_j \in \dot{\bigcup} T_j$ such that $t_{j,o} \in q_j$ and $t_{j,p} \in q_j$ where $q_j \in \dot{\bigcup} T_j$ iff $l_{j,o} = l_{j,p} : l_{j,o}, l_{j,p} \in S_j$ and $E_j(t_{j,o}) = E_j(t_{j,p})$ (transitions with the same state source and triggers are not distinguished);
- *every-interaction* – every $(d_{i,k}, t_{j,l})$ interaction point coverable via $(o_{i,k}, i_{j,l})$ is covered by at least one valid test trace.

The independence of sender and receiver options overall results in $3 \times 3 = 9$ selectable system-level interaction coverage criteria. Ports, states and transitions can also be specified in the context of which interaction sendings and receptions are considered (*port-*, *state-* and *transition-include*, *transition-exclude*).

In order to demonstrate the difference between the various options, consider the *Door.close* event in the elevator example from the *cabin controller*'s (that is, the sender's) point of view. In the case of *events* criterion, the coverage of this event is sufficient regardless of the raising transition (or state in other models), that is, any of the transitions raising this event, the one leaving *Idle*, or entering *TravellingDown* or *TravellingUp* can be covered. In the case of *every-interaction*, every aforementioned transition must be covered. *States-and-events* is similar to *every-interaction*, however, transitions leaving the same state and triggered by the same event are not distinguished. Thus, if state *Idle* had other outgoing transitions triggered by *Door.close*, covering at least one of them would be sufficient. The coverage criteria from the receiver's point of view are analogous to that of

the sender with the exception that only transitions can participate in the reception/processing of events, states cannot.

Dataflow coverage

In the case of *dataflow coverage*, we distinguish between *internal dataflow* in a single atomic component A_i and *interaction dataflow* between A_i and A_j atomic components where parameterized output $o_{i,k} \in O_i$ is connected to parameterized input $i_{j,l} \in I_j$. The only difference is that in the case of *internal dataflow*, variable declarations are considered, whereas in the case of *interaction dataflow*, parameter declarations (which can be considered as write-read variables from the senders and receiver's point of view) of sent and received events are considered as the target of the *definition* and *use* statements.

The coverage criteria build on the following concepts:

- *def d* denotes the definition of the d (parameter or variable) declaration (by assigning a value to it) in the context of a transition (denoted by *def (d,* $t_{i,k}$*)* where $t_{i,k} \in T_i$) or the entry/exit action of a state (denoted by *def (d,* $s_{i,l}$*)* where $s_{i,l} \in S_i$).
- *use d* denotes the use of the d declaration in the context of a transition denoted by *use (d,* $t_{i,k}$*)* (where $t_{i,k} \in T_i$ for internal and $t_{i,k} \in T_j$ for interaction dataflow), or the entry/exit action of a state denoted by *use (d,* $s_{i,l}$*)* (where $s_{i,l} \in S_i$ for internal and $s_{i,l} \in S_j$ for interaction dataflow). A declaration can be used in a predicate (for a decision), denoted by *p-use d*, or in a computation (for a definition), denoted by *c-use d*.
- A *def-clear d path* denotes a computation path without defining the d declaration, that is, the $P = (s_v, t_{v+1}, s_{v+1}, \cdots, t_w, s_w)$ subsequence of a valid execution trace $X = (s_0, t_1, s_1, \cdots, t_j, s_j, \cdots, t_m, s_m)$ where $0 \leq v < m$ and $v < w \leq m$, and $\nexists$ *def (d,* $t_{i,k}$*)* or *def (d,* $s_{j,l}$*)* where $t_{i,k}$ or $s_{j,l}$ is covered by P.
  - A P *def-clear d path* starts from a *def d* statement if $\exists$ *def (d,* $s_{i,k}$*)* where $s_{i,k} \in s_{v-1}$ or *def (d,* $t_{j,l}$*)* where $t_{j,l} \in t_v$.
  - A P *def-clear d path* covers the *use d* statement if $\exists$ *use (d,* $s_{i,k}$*)* where $s_{i,k}$ is covered by P or *use (d,* $t_{j,l}$*)* where $t_{j,l}$ is covered by P.

The tool supports the following options for dataflow coverage criteria:

- *all-def* – for every d declaration, starting from every *def d* statement, at least one *use d* statement (if exists) is covered by at least one *def-clear d* path;
- *all-c-use/all-p-use* – for every d declaration, starting from every *def d* statement, every *c-use d*/*p-use d* statement is covered by at least one *def-clear d path*;
- *all-use* – for every d declaration, starting from every *def d* statement, every *use d* statement is covered by at least one *def-clear d path*.

In the case of this coverage criteria, variables (internal dataflow) or ports (interaction dataflow) can be specified in the context of which the *def* and *use* statements are considered (*port-* and *variable-include*, *variable-exclude*).

4.2 Annotating composite models

In order to support the coverage criteria specified in Sect. 4.1, the composite model is processed (Sect. 4.2.1) and prepared to serve as a basis for the formulation of test targets (Sect. 4.2.2).

4.2.1 Annotating model elements

The composite model gets annotated in a (from a composition semantics point of view) *semantic-independent way*, that is, where necessary, auxiliary elements are inserted into the model according to the selected coverage criteria (see Sect. A2 in the Appendix for details). These auxiliary elements enable the explicit formulation of model element coverage, i.e., the coverage of transitions, declaration definitions and uses (dataflow) as well as behavior-related concepts, such as interactions.

The merit of these annotation methods is that they are *independent* of the composition mode (execution and interaction semantics) used in the composite model and work both in asynchronous and synchronous Gamma models. This trait also holds for interactions: the different characteristics of component interactions in different composition modes can be ignored during annotation as the presented solution (sending/receiving identifiers in event parameters) is universal and works in synchronous-reactive, cascade and asynchronous-reactive models. The characteristics of the composition modes are handled by the transformations

```
pattern interactions(sender : RaiseEventAction, outPort :
  Port, event : Event, inPort : Port, rec : Transition) {
 // Events raised by transitions
 find eventRaisesOfTrans(sender, outPort, event, inPort);
 find transitionTriggers(rec, inPort, event);
} or {
 // Events raised by entry actions
 find eventRaisesOfEntryActions(sender, ...);
 find transitionTriggers(rec, inPort, event);
} or {
 // Events raised by exit actions
 find eventRaisesOfExitActions(sender, ...);
 find transitionTriggers(rec, inPort, event);
}
```

Fig. 6 VIATRA graph pattern specifying interaction points between potential senders and receivers.

deriving analysis models (see Sect. 5.2) according to the corresponding execution and interaction semantics. Consequently, different test cases (generated for the used composition mode) are generated even when the atomic components and selected coverage criteria are identical.

4.2.2 Exploring elements for annotation

As a key aspect, the approach utilizes model queries for the specification and automatic exploration of model elements for annotation. Model queries are defined in the query language of VIATRA[9] [24], which facilitates the high-level, declarative description of complex interrelations between model elements in the form of graph patterns. Queries are treated as input for the annotation process, making the process configurable and easily extensible with additional criteria. The snippet in Fig. 6 describes a graph pattern specifying interaction points between potential senders and receivers connected via channels.

Model elements expected to be returned by model queries are defined in the parameter list of the graph pattern. In the case of *interactions*, the action that raises the event (*sender*), the port of the atomic component via which the event is raised (*outPort*), the raised event (*event*), the port of the atomic component via which the event is received (*inPort*), and the transition triggered by the event (*rec*) are returned. The interconnections of these elements are specified in the pattern body, which consists of three *disjunctive blocks*, that is, the result set of the query will be the union of the result sets of these three blocks. In a single block, the specifications are in an And-relation, that is, every specification (reused pattern in the example) must hold in the result set.

The first block describes interconnections originating from the event raise actions of transitions (by reusing auxiliary patterns), whereas the other two blocks cover events raised by entry and exit actions of states. In the first block, pattern *eventRaisesOfTrans* specifies events (*event*) raised by a transition action via a certain port (*outPort*) of an atomic component connected via channels to the receiving port (*inPort*) of another atomic component. Pattern *transitionTriggers* specifies transitions triggered by events (*event*) of the ports (*inPort*) specified by the *eventRaisesOfTrans* pattern. The blocks for entry and exit actions are analogous to the first block, the only difference is the first reused pattern.

4.3 Generating properties

With the composite model being annotated, the next step is to automatically generate properties in GPL according to criteria selected in GGL or GPPL, describing the coverage of the annotated elements (injected auxiliary elements) serving as test targets [19]. Generally, GPL supports the definition of arbitrary CTL* expressions in the context of the (annotated) composite model, although, only a small subset of the language is used during test generation. Nevertheless, GPL has an important role in the framework in providing a common language for property definition and their mapping into the different property languages of integrated model checker back-ends.

During test generation, every GPL property describes a reachability criterion, using the E (exists) quantifier and F (eventually) temporal operator pair, describing in CTL* that *"there exists a path, along which, eventually"* a certain property (the coverage of a test target) holds. In order to facilitate understandability, comments are automatically generated and assigned to properties, describing the coverable criteria in a human-readable way.

State and out-event coverage

In the case of *state* and *out-event* coverage, the state and event elements of the components are explicitly referred to (using the naming convention applied in the transformation to the model checker

[9] https://eclipse.org/viatra/

input) in the generated properties using the **state** and **out-event** keywords, respectively.

```
@("Covering state 'Open'  in region 'main' of '
     cabinDoorControl'")
E F (state cabinDoorControl.main.Open)
@("Covering out event 'close' of port 'Door' of '
     cabinControl'")
E F (out-event cabinControl.Door.close)
```

Transition coverage

In the case of *transition* coverage, the injected auxiliary boolean variables (i.e., their state where they are set to true) are referred to from the generated properties using the **var** keyword. Note that variable identifiers must be unique, and thus, they may be hard to interpret, which can be mitigated by comments linking the referred variable to the coverable transition.

```
@("Covering the transition between states
 'Open' and 'Closed' in the 'main' region of '
     cabinDoorControl'")
E F (var cabinDoorControl.main_open__main_closed)
```

Transition-pair coverage

In the case of *transition-pair* coverage, the two integer variables generated for states with incoming and outgoing transitions are referred to in the generated properties. The *previousId* and *actualId* variables are specified to contain the identifiers of every incoming and outgoing transition of the specific state, in every possible combination.

```
@("Covering the transition pair between
 states 'TravellingUp' and 'Idle' (id 1)
 and loop transition of state 'Idle' (id
 2) in region 'main' of 'cabinControl'")
E F (var cabinControl.main_previousId == 1
 and var cabinControl.main_actualId == 2)
```

Interaction coverage

In the case of *interaction* coverage, the two integer variables generated for regions containing receiver transitions are referred to in the generated properties. The *senderId* variable is specified to contain a potential sender identifier, and the *receiverId* variable is specified to contain a receiver identifier in accordance with the interaction points identified by using model queries. Note the automatically generated comments as the manual linking of identifiers to the interaction participants is cumbersome for humans.

```
@("Covering the interaction between the
 'open' raising action of the loop
 transition of state 'Idle' (id 3) of
 'cabinControl' and the transition between
 the 'Closed' and 'Open' states (id 4) of
 'cabinDoorControl'")
E F (var cabinDoorControl.senderId == 3
 and var cabinDoorControl.receiverId == 4)
```

Dataflow coverage

In the case of *dataflow* coverage (both for *internal* and *interaction dataflow*), the integer variables generated for *use d* statements are referred to in the generated properties: for each included *d* variable, for each *def d* statement, "at least one" or "every" *use d* statement is specified to be covered by a *def-clear d path* according to the selected coverage criterion. In the case of *interaction dataflow*, the possible dataflow through interactions are explored using the model query presented in Fig. 6. Note that below only *all-use* statements are described; the approach is identical for *c-use/p-use* statements and differ only in selecting ids for use d variables.

```
@("All-def: from every def d statement, at least one
     use d statement is covered by at least one def-
     clear d path")
E F (var useD == 1 or ... or var useD == n)
@("All-use: from every def d statement, every use d
     statement is covered by at least one def-clear d
     path")
E F (var useD == 1) ...
E F (var useD == n)
```

5 Model processing

After annotating models and generating properties according to the selected coverage criteria (see Sect. 4), composite models are further processed by *(i)* executing *model reduction* and *slicing* algorithms (Sect. 5.1) to reduce their state space, and *(ii)* transforming composite models into *analysis models* (Sect. 5.2) to enable model checking for the coverage of test targets (abstract test generation process). During and after abstract test generation, *optimizations* based on the generated abstract tests are carried out (Sect. 5.3) to decrease the time of the process as well as the size of the resulting test sets. Note that these transformation techniques are not test generation specific and have been integrated into the general Gamma workflow, too.

5.1 Model reduction & model slicing

The goal of both model reduction and model slicing is to remove "unnecessary" elements from the composite models in order to speed up the model checking (and thus, the end-to-end test generation) process. The difference is that model reduction (Sect. 5.1.1) is *general* and ignores potential verifiable properties whereas model slicing (Sect. 5.1.2) is carried out on the basis of *verifiable properties* (and thus, the selected coverage criteria) and removes model elements that are unnecessary in a *concrete* model checking run.

5.1.1 Model reduction

Model reduction depends only on the underlying model and removes unused and inoperative elements from the statechart models, such as unused variables and events, unfireable transitions, unreachable states and removable regions. The definition of these concepts are as follows.

- A variable is *unused*, if there is no definition or use statement in the statechart model that refers to it.
- An event is *unused*, if there is no transition triggered by it (input event) or there is no raise event action of a transition or state that raises it (output event) in the statechart model.
- A transition is *unfireable*, if *(i)* it is triggered by an input event that cannot be received by the statechart model (e.g., due to the lack of channels, detailed in a graph pattern presented in Fig. 7) or *(ii)* its guard evaluates to constant false (e.g., due to a model parameter value).
- A state is *unreachable*, if it has no incoming transitions.
- A region is *removable*, if it *(i)* contains no states or *(ii)* contains a single entry node and a single simple (non-composite) state without entry or exit actions with an empty transition between the entry node and the state.

The specification and exploration of such elements are supported by model queries. For example, unfireable transitions and unreachable states are explored using the declarative graph patterns specified in Figs. 7 and 8.

A transition is specified *unfireable* by the graph pattern (see Fig. 7) if it is triggered by an event (reused *transitionTriggers* pattern in the body of the *unfireableTransitions* pattern) that the containing atomic component cannot receive due to the absence of event raise actions in connected components or the absence of connections altogether (negated use of the *triggerableTransition* pattern in the body). As transitions can be triggered by a *combination* of events, the *transitionTriggers* pattern specifies events that can trigger a transition by using the *transitive closure* construct with the *binaryTriggerToTrigger* pattern (note the + operator in the pattern call) to iterate through potential intermediate binary triggers.

```
pattern unfireableTransitions(transition : Transition) {
 find transitionTriggers(transition, port, event);
 neg find triggerableTransitions(transition, port, event);
}
pattern transitionTriggers(transition : Transition, port :
      Port, event : Event) {
 Transition.trigger(transition, trigger);
 find binaryTriggerToTrigger+(trigger, descendantTrigger);
 ...
}
pattern binaryTriggerToTrigger(parent : BinaryTrigger,
     child : Trigger) {
 BinaryTrigger.leftOperand(parent, child);
 ...
} or {
 BinaryTrigger.rightOperand(parent, child);
 ...
}
```

Fig. 7 VIATRA graph patterns specifying conditions for unfireable transitions.

A state is specified *unreachable* by the graph pattern (see Fig. 8) if there is no transition in the model (negated use of the *transition* pattern is the body of the *unreachableState* pattern) that enters the particular state.

Graph patterns are tailored to the declarative definition of interconnections between elements, and thus, are less suitable for interpreting arithmetic and boolean expressions. Nevertheless, such a function would prove beneficial in filtering unnecessary results from the result sets, e.g., by checking the argument list of raise event actions and transition guards to filter out impossible interaction points. To this end, we employ postprocessing on the returned result sets using imperative code for expression interpretation.

For example, in the case of the *interactions* pattern presented in Fig. 6, interconnections that can definitely *not result in valid interactions* can be discarded by interpreting the guards of receiver transitions, and thus, reducing unnecessary model checking efforts during test generation. The tool

```
pattern transition(source: StateNode, target: StateNode) {
 Transition.source(transition, source);
 Transition.target(transition, target);
}
pattern unreachableState(state : State) {
 neg find transition(_, state);
}
```

Fig. 8 VIATRA graph patterns specifying conditions for unreachable states.

does just that: in the case of parameterized events, the argument list of the sender is checked, and the guard of the potential receiver transition is evaluated with the corresponding values using imperative code. The specific interconnection is discarded if the guard evaluates to false. Similarly, in the case of unfireable transitions (Fig. 7), the result set is further extended by interpreting their guards using imperative code: transitions whose guards evaluate to constant false, e.g., due to parameter values, are also specified unfireable.

In general, model queries facilitate the incremental and thus, rapid evaluation of changes in the underlying model, e.g., in VIATRA, the effects in result sets caused by removing model elements can be evaluated in almost zero time. This trait can prove profitable during the iterative exploration and retrieval of removable regions and unreachable states after removing transitions as the removal of elements affects the result set of other model queries.

5.1.2 Model slicing

Model slicing depends on properties to remove elements ignorable in a concrete model checking run. The technique removes unnecessary written-only variable declarations and variable definitions along with potentially unnecessary output events and event raisings. The definitions of the concepts are as follows.

- A *written-only variable declaration* is *unnecessary*, if there is no *use* statement in the statechart model or variable reference in the verifiable properties that refers to it.
- A *variable definition* is *unnecessary*, if it defines an unnecessary written-only variable.
- An *output event* is *unnecessary*, if there is no transition in connected statechart models that are triggered by it or out-event reference in the verifiable properties that refers to it.

```
pattern writtenOnlyVariables(var : VariableDeclaration) {
 find writtenVariables(var);
 neg find readVariables(var);
}
pattern writtenVariables(var : VariableDeclaration) {
 find lhsOfAssignments(ref);
 ReferenceExpression.declaration(ref, var);
}
pattern readVariables(var : VariableDeclaration) {
 neg find lhsOfAssignments(ref);
 ReferenceExpression.declaration(ref, var);
}
pattern lhsOfAssignments(ref : ReferenceExpression) {
 AssignmentStatement.lhs(ass, ref);
}
pattern referenecesInProperties(v : VariableDeclaration) {
 find expressionsInProperties(exp);
 ReferenceExpression.declaration(exp, v);
}
pattern unnecessaryVariables(var : VariableDeclaration) {
 find writtenOnlyVariables(var);
 neg find referenecesInProperties(var);
}
```

Fig. 9 VIATRA graph patterns specifying conditions for slicing.

- An *event raising* is *unnecessary*, if it raises an unnecessary output event.

As a special setting, a *system output event* (that is an output event led out of the system model) can be specified *unnecessary* explicitly before the test generation procedure (in the gen-model configuration file) if output event references in the generated tests are not required. Note that this can be undesirable in most cases, hence the necessity for the explicit specification.

Model slicing utilizes graph patterns in a similar way as model reduction in order to explore "unnecessary" elements in the composite model; the difference is that these graph patterns also take the generated properties into account.

For example, a variable declaration in a statechart model is specified *written-only* by the graph pattern (see Fig. 9) if it is written by an assignment statement (specified by the *writtenVariables* pattern), but never referred to in other expressions (negated use of *readVariables*). Variables referred to in properties (*variablesInProperties*) are specified by reusing the *expressionsInProperties* pattern, which explores all contained expressions inside property specifications using transitive closure constructs; from these expressions, variable references are selected. Unnecessary variables are written-only variables that are not referred to in properties (*unnecessaryVariables*). Unnecessary output events are specified in a very similar way.

In the elevator example, if we explicitly specify the coverage of the transition between states *Open* and *Closed* with the property *E F (var cabinDoorControl.main_open__main_closed)*, the boolean variable generated for the other transition with name *main_closed__main_open* is considered an unnecessary written-only variable whose value is irrelevant in the concrete model checking run, and thus, gets removed from the model during slicing.

In general, the above examples in model reduction and slicing highlight the expressive power of model queries and graph patterns: the negated reuse of declarative patterns and the transitive closure constructs support the exploration of elements (possibly via the iterative application of different graph patterns) to which certain conditions do *not* hold, e.g., a certain event via a port is never received or there is no transition entering a certain state. The exploration of such interconnections may require the traversal of the entire model, which, in this way, is automatically taken care of by the query engine in an efficient way.

By removing unnecessary transitions, states, regions and variables, model reduction and model slicing can significantly reduce the state space of the resulting model, and thus, the time needed for model checking runs and test generation in total. The results are presented in Sect. 6.

5.2 Transforming models into target models for test generation

Gamma currently allows deriving three fundamentally different target (analysis) model types from composite models to complement each other and support different types of inputs (models with different characteristics, e.g., timed and non-timed models) as well as different composition modes. The following paragraphs overview the *DU – direct UPPAAL timed automata model*, the *DX – direct XSTS model* and the *XU – XSTS-based UPPAAL timed automata model* transformations.

DU models are derived by directly mapping composite models into the timed automata formalism of UPPAAL [25]. The resulting model forms a network of automata, where atomic components (modeled as separate automata) communicate via shared variables and synchronization constructs supported by the language. System execution is conducted by auxiliary scheduler automata, controlling the execution of instantiated components. Currently, this is the only transformation in Gamma supporting asynchronous models.

In the *DX* transformation, the composite model is mapped into a transition system formalism, namely the *eXtended Symbolic Transition System* (XSTS) [26], supported by Theta. XSTS is an analysis language with low-level constructs that serves as a common representation of composite models, facilitating the integration of various model checker back-ends into the framework. An XSTS model consists of a set of variables that describe the state of the system (including interactions between components) and a set of transitions specifying possible changes from one system state to another. Transitions can use atomic actions, such as assignment actions or assume actions (assumptions that must hold in order to execute specific action branches), which can be reused to construct composite actions, such as sequential actions (blocks) or deterministic (if-else) and nondeterministic choices.

XU models are derived by transforming 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 that are based on *control locations* and resemble the structure of statecharts, these models can be considered as *control flow automata* (CFA) [27] 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.

Even though the structure of the derived target models is fundamentally different, their behavior fully conforms to the formal semantics of the Gamma languages in terms of atomic component behavior (GSL semantics), interactions (GCL semantics) and timed behavior (GSL and GCL semantics). The formalized Gamma semantics is published in [11]. The transformation rules of mapping Gamma models into semantically equivalent UPPAAL automata are presented in [25] while documentation of XSTS can be found in [26]; the detailed descriptions of the target analysis languages and the model transformations are out of the scope of this paper.

Utilizing the traits of these different target model types, Gamma can support the modeling

and verification of high-level collaborating statecharts with fundamentally different constructs and methods in a flexible way as different model types and composition modes may be best handled with different analysis models (timed automata, transition systems and CFA) as well as model checkers and techniques, i.e., UPPAAL using explicit-state techniques and Theta with CEGAR-based abstraction techniques. The extensive evaluation of the tool's verification capabilities, considering the above aspects, is presented in Sect. 6.

5.3 Optimizing abstract test generation

The tool employs two optimization algorithms to reduce the test generation time and the number of generated tests during abstract test generation.

After generating a *new abstract test case*, the first algorithm iterates through the still unchecked properties and checks whether the test case also covers some test requirements specified by the *remaining* properties [19]. Such properties get discarded (they are not examined in later model checking runs) as the test requirements, the coverage of which they specify, are already covered by the specific generated test case.

After checking *every property*, the second algorithm is applied, which searches for test cases in the generated test set that are *prefixes* of other test cases [28, 29]. Note that such test cases can exist even when the first algorithm is applied as the order in which the properties are checked is random. Such test cases do not contribute to the coverage of additional criteria (all covered elements are also covered by other test cases), and thus, can be discarded to further reduce the generated test set. Altogether, these algorithms are lightweight even though they can significantly decrease the generation time as well as the number of generated test cases in most circumstances (see Sect. 6 for results).

Note that the outcome of the first optimization algorithm depends on the order of checking the properties. In turn, the effects of the second algorithm is always deterministic given a set of test cases, i.e., multiple executions return the same resulting test set if the model checker always returns the same traces.

6 Practical experiences

This section presents practical experiences about our Gamma test generator tool. Section 6.1 demonstrates the *usability* of the tool from a *users' point of view* in the context of practical examples from the industry. Then, building on the models and configurations introduced in Sect. 6.1, Sect. 6.2 presents *more detailed results* about the test generation process and evaluates its *efficiency* considering the different composition modes, built-in model processing and optimization techniques (model reduction, slicing and test optimization) as well as analysis models and model checkers. Section 6.3 presents conclusions and threats to validity with respect to our evaluations.

6.1 Evaluation of tool usability

We have examined related work (see Sect. 7) for models on which we could demonstrate the usability of our tool but encountered difficulties: the used models are either *(i)* small in size, *(ii)* method-specific (constructed for their approach in an ad hoc way) or *(iii)* commercial models not published in their entirety. These findings correlate with the conclusion of [19], according to which there is a lack of documented empirical experience with model checker based testing, hindering the selection of preferred techniques for given test scenarios. For example, [30] presents a survey regarding the scalability of model checker based solutions and comes to promising conclusions, but a later study [31] shows that the considered applications have some particularities, questioning the representativeness of the results. Nevertheless, we could select a single component that was used in multiple works but in its case neither integration testing nor the effects of our model reduction techniques could be presented besides the effects of our model checker algorithm choice (we overview the results in Sect. 6.2.3). Consequently, we evaluate our tool on three system models (previously unused in related work) received from our industrial partners Prolan and NASA, while also making them publicly available.

In the following, this section presents test generation in our tool from a users' point of view for three system models, namely the *signaller subsystem* (Sect. 6.1.1), the *simple space mission* (Sect. 6.1.2) and the *railway path locking topology*

Table 1 Number of *states*, *transitions* and *variables* (including timeout variables) in the system models used for evaluation.

	#State	#Transition	#Variable
Signaller	26	97	5
Space mission	9	19	8
Railway path	110	407	61

(Sect. 6.1.3) models. Statistics of the models are summarized in Table 1; for additional details, we direct the reader to Sect. A3 of the Appendix.

To highlight the applicability of our tool in the case of different system models and composition modes, we specify relevant coverage criteria for every examined system model and generate tests using all supported analysis models while employing every possible model reduction and slicing as well as test optimization technique. We present test generation time (most efficient analysis model and model checker result) in addition to the necessary test steps after optimization for the coverage of test targets in every composite model variant.

We *jointly* measure the process time for *model transformation* and *test generation* and repeat the measurements $1 + 5$ times for each configuration while not measuring the first execution to mitigate the potentially distorting effects of the JIT compiler and then calculating median values.[10] In the experiment, there are no manual steps, the whole workflow in Gamma is automatic. During model checking, we use the default options both for UPPAAL and Theta models.

Regarding results, we expect that the nondeterministic nature of asynchronous models with potential interleavings (that are hard to test in general) will pose a great challenge for our test generation tool (that is, the UPPAAL model checker back-end), which can result in very long execution times or even the inability to cover certain properties, e.g., due to too much required memory. In turn, we do not expect that synchronous-reactive and cascade models will pose significant difficulties for our tool. We also expect that the cascade models will be easier to manage due to the specified execution semantics and employed modeling constructs.

[10]The measurements were run on the following configuration: *Intel Core (TM) i7-4700MQ @ 2.40GHz, DDR3 16GB @ 1.6GHz, SSD 500GB*.

6.1.1 Test generation for a safety-critical signaller subsystem

The *signaller subsystem* comprises models received from Prolan, an industrial partner developing railway traffic control systems.[11] The subsystem builds on statechart components (i.e., two antivalence checkers and a signaller – see Figs. 10 and 11) created in different modeling tools with many interaction points among them, providing a good target for *interaction* testing in addition to all kinds of *structural* criteria. The models were designed to be reusable in various contexts, and thus, contain variability points, providing reduction possibilities.[12]

The designers have requested the assessment of various interaction semantics (architectures) in terms of testing and thus, we created three composite model variants in GCL using the synchronous-reactive, cascade and the asynchronous-reactive composition modes. The composite model structure is the same in every variant: the outputs of the antivalence checkers are connected to the inputs of the signaller.

In this system model, we generate test sets aiming at full *state*, *transition*, *transition-pair*, *out-event* coverage as well as *interaction* coverage between the antivalence checkers and the signaller. Regarding *interaction* coverage, we create a test setting where *(i)* both the sender and receiver interaction coverage are set to *events (E)*, *(ii)* both are set to *states-and-events (SE)*, *(iii)* the sender interaction coverage is set to *every-interaction* and the receiver interaction coverage is set to *events (EI-E)* and *(iv)* where both are set to *every-interaction (EI)*.

Model transformation

Annotation for different coverage criteria is carried out according to the rules presented in Sect. 4.2 and A2 in the Appendix. Reduction and slicing are identical for every selected coverage criterion and model variant.

In the signaller model, the *reduction algorithm* removes the transition entering the *Off* state as the trigger event *toggle* cannot be received via the *I* port (as it is not connected anywhere). As

[11]https://www.prolan.hu/en/hirek/PRORIS-H

[12]The models and measurement results can be found at https://github.com/ftsrg/gamma/tree/master/examples/hu.bme.mit.gamma.railway.casestudy/model/COID.

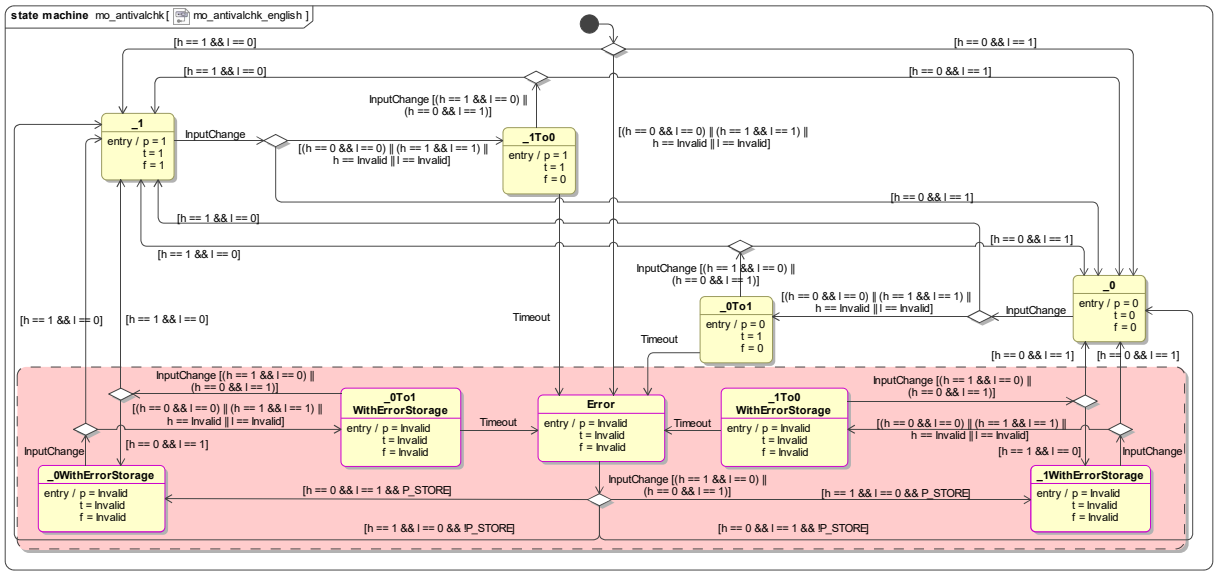

Fig. 10 *Antivalence checker* statechart model.

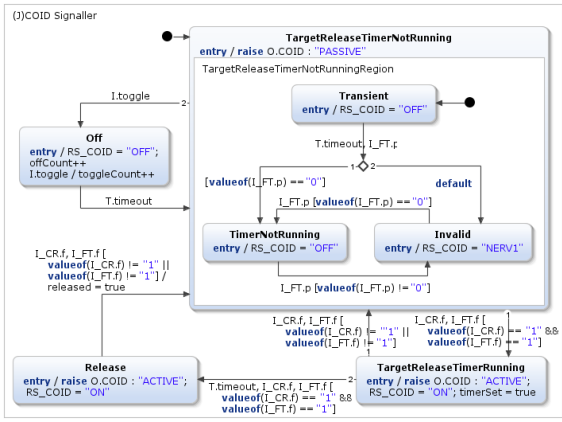

Fig. 11 *Signaller* statechart model.

a result, the *Off* state becomes unreachable, and also gets removed along with its outgoing transition. The *slicing algorithm* removes the *RS_COID*, *released*, *timerSet*, *offCount* and *toggleCount* variables as they are unused or unnecessary written-only variables that are not referred to in the generated properties. In the antivalence checker model, the reduction algorithm removes the transitions leaving the bottommost choice state and entering the *_0* and *_1* states as their guards evaluate to constant false due to the *!P_STORE* subexpression (recall that *P_STORE* is set to true). All in all, one state, 12 transitions and five variables get removed from the system models.

Table 2 *Median test generation time* (left) and the *number of steps* (right) in the test sets generated for the *signaller subsystem* model applying every possible model reduction, slicing and test optimization technique.

	Median generation time (s)/#Generated steps		
Criterion	**Sync**	**Casc**	**Async**
State	3.5/25	3.9/21	189.6/21
Transition	27.4/128	28.6/128	1914.5*/128
Transition-pair	281.6/184	294.1/177	*/*
Out-event	2.0/5	1.1/4	170.1/4
Interaction-E	1.6/8	1.1/6	166.2*/6
Interaction-SE	29.0/33	15.4/27	1945.3*/27
Interaction-EI-E	15.7/33	8.4/21	1013.3*/21
Interaction-EI	28.6/37	15.2/27	1890.6*/27

Test generation

Table 2 presents the median generation time and number of generated steps for each specified coverage criterion (**Criterion**) in the case of the synchronous-reactive (**Sync**), cascade (**Casc**) and asynchronous-reactive (**Async**) composite model variants using the analysis model with the best result. As can be seen, our tool can manage synchronous-reactive and cascade variants relatively well as the greatest test generation time is under 295 seconds (cascade variant with *transition-pair* coverage) while for *state*, *out-event* and *interaction – E* coverage (least resource-intensive criteria) the process finishes under 4 seconds. In turn, in the asynchronous-reactive variant, UPPAAL can return traces to properties

proven satisfiable in the cascade and synchronous-reactive variants. However, according to our initial expectations, for properties proven unsatisfiable in the other variants (and for all properties in the case of the *transition-pair* criterion), UPPAAL cannot provide proof and throws an *out of memory exception* after 250 seconds; hence the * symbol next to certain values as they include time only for the covered properties. Either these properties are indeed unsatisfiable (which we consider plausible based on the results of the other variants) or trace generation is too resource-intensive with such execution and interaction semantics. The size of the generated test sets according to different coverage criteria range from below 10 (*out-event* and *interaction – E*), to between 20 and 40 (*state*, *interaction – SE/EI-E/EI*) and finally, to between 120 and 190 (*transition*, *transition-pair*).

As for the manageability of different coverage criteria, as expectable, the data show that test generation for criteria not requiring extra auxiliary elements during annotation, i.e., *state* and *out-event*, require the least resources. Next, *interaction* coverage criteria, in the order of *E*, *EI-E*, *SE* and *E* (from coarse-grained to more refined criteria) are getting harder to manage – there is an order of magnitude decrease in performance compared to the previous criteria. Note that *SE* and *E* results are very similar due to the fact that they specify the coverage of the same elements apart from the outgoing transitions of state *TargetReleaseTimerRunning* in the *signaller* model (note that in the *antivalence checker* model, every state has a single outgoing transition). Finally, *transition* and *transition-pair* criteria require the most injected auxiliary elements and thus, are generally the hardest to manage: the results in the case of *transition* coverage are close to that of the *interaction – E* criterion; however, in the case of *transition-pair*, there is another order of magnitude decrease in the performance.

6.1.2 Test generation for a simple space mission

The *simple space mission* model comes from the aerospace domain and was initially proposed by NASA in the context of the OpenMBEE[13] framework, a common model repository initiative. The original SysML models were created in MagicDraw in the form of statecharts and activity diagrams which we manually mapped into the GSL and GCL languages of Gamma in [11]. At first glance, the resulting models (see Figs. 12 and 13) do not seem to have a large state space due to the few control locations, but the several timeouts, variable definitions and uses (incrementation, decrementation and guard expressions) can pose a significant challenge for model checkers (especially for symbolic model checking techniques). Therefore, this model is a good target for *dataflow* testing.[14]

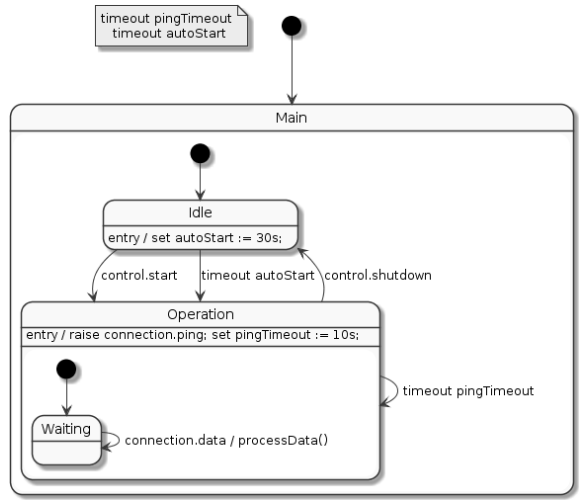

Fig. 12 *Ground station* statechart model with inlined activity diagrams.

In the *simple space mission* model, we generate test sets aiming at *(i) all-def* (a more coarse-grained testing) and *(ii) all-use* (a finer testing) *internal dataflow coverage* within the ground station and the spacecraft components.

Model transformation

As in the previous system model, reduction and slicing phases are identical for every selected coverage criterion and composite model variant in the *simple space mission* model, too. Our reduction and slicing algorithms do not find any reducible elements in this model so the preprocessing procedure includes only the annotation phase with substantive results: integer variables are created in

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

[14] The models and measurement results can be found at https://github.com/ftsrg/gamma/tree/master/examples/hu.bme.mit.jpl.spacemission.casestudy.

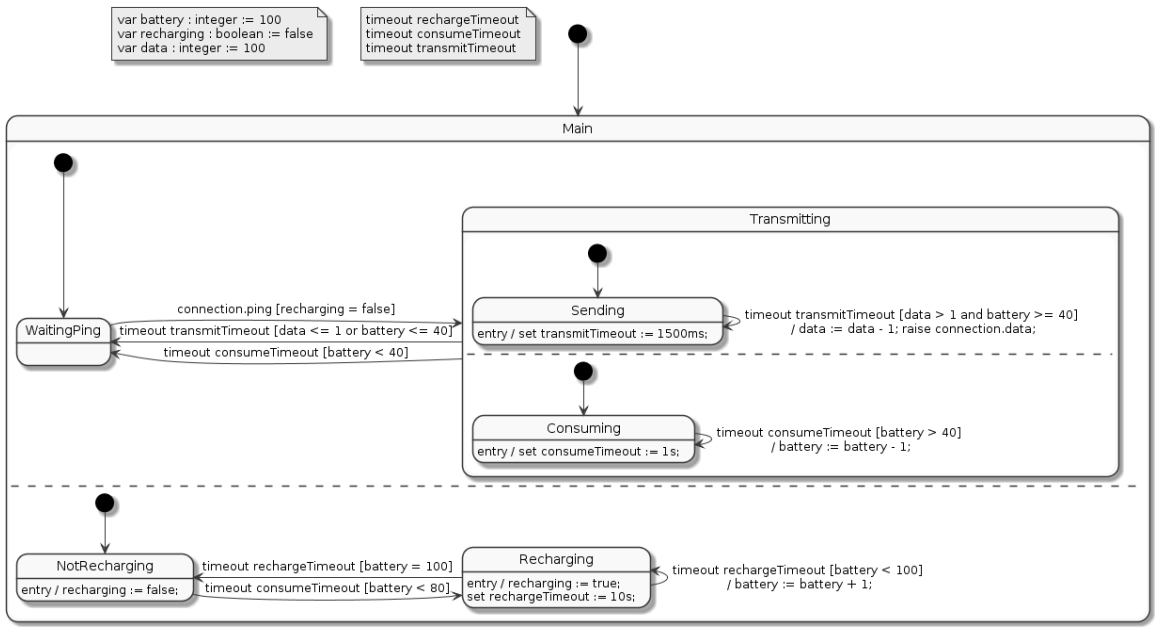

Fig. 13 *Spacecraft* statechart model with inlined activity diagrams.

the spacecraft model (as the ground station model operates only with timeouts). An excerpt of the annotated spacecraft model is shown in Fig. 22 in the Appendix.

Test generation

Table 3 presents the median generation time and number of generated steps for the *all-def* and *all-use* dataflow test set for the *simple space mission* model. The results show that our tool can handle the synchronous-reactive and cascade variants as test generation finished at most in 50 seconds in every case (the slowest being the synchronous-reactive variant with *all-use* coverage – in accordance with our expectations). In turn, similarly to the *signaller subsystem*, UPPAAL can return traces in the asynchronous-reactive variant only for properties proven satisfiable in the cascade and synchronous-reactive variants. In these models, the out of memory exception comes after around 350 seconds; hence the * symbol next to the values. The size of the generated test sets are rather large compared to the other examined system models with around 140 and 950 steps for the *all-def* and *all-use* criteria.

In general, the difference between the *all-def* and *all-use* criteria is the generated properties: in the case of *all-def*, at least one def-use pair has to be covered for each variable whereas *all-use* specifies the coverage of every potential def-use pair. The latter can result in several unsatisfiable properties, highly increasing model checking time and the size of the generated test set as depicted in the table: model checking time increases between five- and tenfold due to the greater number of properties and the number of generated steps increases around sevenfold. Nevertheless, in the synchronous-reactive and cascade variants, testing according to the finer *all-use* criterion still remains

Table 3 *Median test generation time* (left) and the *number of steps* (right) in the test sets generated for the *simple space mission* model applying every possible model reduction, slicing and test optimization technique.

	Median generation time (s)/#Generated steps		
Criterion	**Sync**	**Casc**	**Async**
All-def	10.7/138	5.4/137	271.5*/137
All-use	49.2/947	46.7/949	2913.3*/949

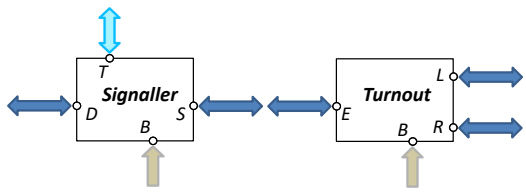

Fig. 14 Schematic descriptions with ports and the direction of communication for the *signaller* and the *turnout* components.

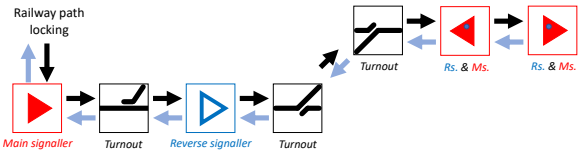

Fig. 15 *Railway path locking topology* consisting of *signaller* and *turnout* components.

manageable, a characteristic which is expected to hold in every model where the model checker can explore the entire state space (with, of course, increased generation time) as these two criteria result in the same model (annotations).

6.1.3 Test generation for a distributed railway path locking topology

The *railway path locking topology* consists of manually created models based on specifications from Prolan. The system comprises two kinds of components (see Fig. 14) designed to function in *multiple roles*, which can be set with the combination of *port connections* (in a topology) and certain *parameters*. The size of the investigated topology (see Fig. 15) is rather large to handle with trivial methods and is a good target for our reduction and slicing techniques to remove parts of the system unrelated (unnecessary) to the selected component roles. Moreover, this topology is a good context to demonstrate the practicability of our approach when targeting the behavior of a *single component* in a complex system and thus, we aim at *transition* and *transition-pair* coverage in this model.[15]

In the *railway path locking topology*, we generate test sets aiming at *(i) transition* (more coarse-grained testing) and *(ii) transition-pair* coverage (finer testing) for the third signaller component counted from the left (with role *reverse & main* – R&M). We selected this component for coverage as it has a more "complex" role in this topology than the first two signallers and communicates via all its neighboring ports (contrary to the turnouts and other *R&M* signaller) which overall can result in more transitions (a greater state space) to cover.

[15]The models and measurement results can be found at https://github.com/ftsrg/gamma/tree/master/examples/hu.bme.mit.gamma.railway.casestudy/model/RailwayPathLocking.

Model transformation

In the annotation phase, a resettable boolean variable is created for every coverable transition in the selected *R&M* signaller component in the case of transition coverage, and a pair of integer variables (*previousId* and *actualId*) is created for each state with at least one incoming and an outgoing transition (see Sect. 4.2 and A2 in the Appendix for details).

Similarly to the previous system models, the reduction and slicing phases are identical for every selected coverage criterion and composite model variant. The model reduction and slicing algorithms find 296 removable transitions, 31 removable regions, 143 removable states and 37 removable variables in every composite model variant. As indicated by the number of removed elements, the signaller and turnout models are actually prepared to serve in different roles, most of which can prove unnecessary in concrete topologies and should be removed to facilitate model checking.

Test generation

Table 4 presents the median generation time and number of generated steps for the *transition* and *transition-pair* coverage test set for the synchronous-reactive and cascade composite variants of the *railway path locking topology* model – our tool cannot generate tests for the asynchronous-reactive model variant. The data show that both test generation time and the size of the generated test set remain manageable with 30 seconds and 20 steps being the greatest values for this model (synchronous-reactive variant with *transition-pair* coverage) – a result made possible by the model reduction and slicing techniques.

In general, as highlighted in the table, the modeling constructs describing the execution of incoming-outgoing transition pairs can greatly increase the state space compared to the use of single resettable boolean variables: the increase

Table 4 *Median test generation time* (left) and the *number of steps* (right) in the test sets generated for the *railway path locking topology* model applying every possible model reduction, slicing and test optimization technique.

Median generation time (s)/#Generated steps		
Criterion	**Sync**	**Casc**
Transition	15.2/18	5.2/6
Transition-pair	29.6/20	14.5/9

is between two- and threefold depending on the model variant. Consequently, we can expect that with the increased number of states, testing according to a *transition-pair* criterion would scale far worse than for *transition* coverage and in general, any other supported criterion.

6.2 Evaluation of model processing and optimization techniques

This section presents a more thorough, technology-oriented evaluation of our Gamma test generator tool based on the models and coverage criteria defined in the previous section. It focuses on the effects of the model processing and annotation techniques as well as the traits of the different composition modes besides the supported analysis models and model checker back-ends. In order to drive the evaluation, we formulated the following questions (Q) that we aim to answer based on an extensive measurement campaign and our previous experiences.

1. How do model processing techniques (model reduction, slicing and test optimization) affect test generation time, and independently, model mapping and model checking time?
2. How do different composition modes (execution and interaction semantics) affect test generation time? Can the tool handle asynchronous-reactive composite models?
3. How do different model checking configurations (analysis models and model checker back-ends using different techniques) affect test generation time?

Every question focuses on test generation time as the main problem of model checker-based testing is considered to be *performance* [19] and thus, we evaluate our test generation approach in this regard. To highlight the applicability of different analysis models and model checkers and demonstrate the cost and efficiency of our model annotation and optimization algorithms, we organize our measurement campaign according to the following points.

- We select two coverage criteria, a more coarse-grained one and a finer one of the same "criterion family" for the models presented in Sect. 6.1.
- We define transformation configurations that differ in the composition modes (synchronous-reactive, cascade and asynchronous) and analysis models (*DU*, *DX* and *XU*).
- We extend these configurations by adding an option whether model reduction and slicing are employed.
- We extend these configurations by adding an option whether test optimization is employed.

Contrary to Sect. 6.1, we *independently* measure time for *model transformation* and *test generation* in each configuration and repeat the measurements 1+5 times. We also present the number of generated properties (test targets) and actually satisfiable properties as well as the necessary steps for their coverage in different composite model variants with and without test optimization.

After carrying out the measurements, we found out that the trends and conclusions were similar for the more coarse-grained and the finer test coverage criteria of the same criterion family in the particular system models. Thus, the rest of the section presents detailed measurement results only for the finer (more resource-intensive) criteria and includes only high-level findings and statistics (e.g., median values) for the more coarse-grained criteria. Nevertheless, the detailed measurement results for the more coarse-grained criteria can also be found in the aforementioned public repository.

Signaller subsystem

In the case of the *signaller subsystem*, we selected the *interaction – EI-E* and *EI* coverage criteria for further evaluation as the models provide a good basis for examining the traits of our interaction coverage techniques.

Table 5 presents the median time to transform the composite model variants (synchronous-reactive, cascade and asynchronous-reactive) targeting *EI interaction* coverage into a *DU*, *DX* and *XU* model (**M**), with or without employing reduction and slicing techniques (**R&S**). Recall that currently only the *DU* transformation supports asynchronous models, hence the empty fields in the lines of *DX* and *XU* (and also in the case of other system models in subsequent sections).

During annotation, the model queries identify altogether 27 interaction points in the case of the *EI-E* criterion between the two antivalence

Table 5 *Median time* (ms) to map the *signaller subsystem* models into analysis models.

Median transformation time (ms)				
M	**R&S**	**Sync**	**Casc**	**Async**
DU	✗	557	549	585
	✓	612	620	631
DX	✗	1368	1289	–
	✓	1196	1205	–
XU	✗	1432	1373	–
	✓	1292	1259	–

checkers and the signaller in all composite model variants, which would be 45 without interpreting the guards of the receiver transitions. In the case of *EI* coverage, 49 potential interaction points are identified in all composite model variants, which would be 90 without guard interpretation.

The model checkers exhibit that 11 interaction points for the *EI-E* criterion and 15 for *EI* coverage can be covered in every model variant.

Table 6 presents the median generation time and number of generated steps in the case of the *EI* interaction criterion, based on the *DU*, *DX* and *XU* models (**M**) generated with or without employing the reduction and slicing algorithms (**R&S**) or the test optimization algorithms (**T**). As discussed previously, UPPAAL can return traces in the asynchronous-reactive variant only for properties proven satisfiable in the cascade and synchronous-reactive variant. Moreover, in *unreduced models*, Theta cannot return traces for most properties before running out of memory, hence the lack of the rows for *DX* in the tables.

Table 6 *Median time* (s) to generate (top) and the *number of steps* (bottom) in the **every interaction** coverage test set for the *signaller subsystem* composite model variants.

Median generation time (s)					
M	**R&S**	**T**	**Sync**	**Casc**	**Async**
DU	✗	✗	80.0	36.9	–
		✓	77.5	35.4	–
	✓	✗	32.7	17.9	1991.1*
		✓	28.0	14.6	1890.0*
DX	✓	✗	359.8	284.0	–
		✓	308.8	205.4	–
XU	✗	✗	296.1	30.7	–
		✓	228.7	28.4	–
	✓	✗	224.4	16.1	–
		✓	199.8	13.5	–

#Generated steps			
T	**Sync**	**Casc**	**Async**
✗	77	60	60
✓	37	27	27
Δ	52%	55%	55%

Simple space mission

In the case of the *simple space mission*, we provide further examination for the *all-def* and *all-use* coverage criteria.

In the annotation phase, the model queries identify altogether five definition statements and 13 use statements. In the case of the *all-def* criterion, five properties are generated (one for each definition), which increases to 23 (one for each def-use pair) in the case of the *all-use* criterion.

Table 7 presents the median time to transform the composite model variants targeting *all-use* dataflow coverage into analysis models.

The model checker back-ends exhibit that in the case of the *all-def* criterion, four out of five properties can be satisfied (a def-use path for *recharging* is not coverable) whereas in the case of the *all-use* criterion, 13 out of the 23 properties can be satisfied in every composite model variant.

Table 8 presents the median generation time and number of generated steps for the *all-use* dataflow criterion in the case of the different composite model variants. As discussed previously, UPPAAL can return traces in the asynchronous-reactive variant only for properties proven satisfiable in the cascade and synchronous-reactive variant. Moreover, the large number of incrementations and decrementations in the models poses a challenge too great for the symbolic techniques of Theta as it cannot return results for most of the properties before running out of memory, hence the lack of the *DX* rows in the table.

Railway path locking topology

In the case of the *railway path locking topology*, we further examine the *transition* and *transition-pair* coverage criteria.

Table 7 *Median time* (ms) to map the *simple space mission* models into analysis models.

Median transformation time (ms)				
M	**R&S**	**Sync**	**Casc**	**Async**
DU	✗	331	406	473
	✓	382	402	517
DX	✗	181	184	–
	✓	237	244	–
XU	✗	202	201	–
	✓	257	269	–

Table 8 *Median time* (s) to generate (top) and the *number of steps* (bottom) in the **all-use** coverage test set for the *simple space mission* composite model variants.

Median generation time (s)

M	T	Sync	Casc	Async
DU	✗	305.4	256.3	3142.0*
	✓	282.6	235.3	2912.8*
XU	✗	57.6	53.0	–
	✓	48.8	46.3	–

#Generated steps

T	Sync	Casc	Async
✗	1621	1608	1608
✓	947	949	949
Δ	42%	41%	41%

In the case of the *transition* coverage criterion, the model queries identify six coverable transitions whereas in the case of the *transition-pair* coverage criterion, five input-output transition pairs are explored related to two different states in every composite model variant.

Table 9 presents the median time to transform the composite model variants targeting *transition-pair* coverage into analysis models.

The model checkers exhibit that in the case of *transition* coverage, five out of the six transitions whereas in the case of *transition-pair* coverage, three out of the five transition-pairs can be covered in the synchronous-reactive and cascade variants. UPPAAL cannot return any traces due to out of memory exceptions in the asynchronous-reactive variant, hence the missing column.

Table 10 presents the median generation time and number of generated steps for the *transition-pair* criterion in the case of the different model variants. Note that without employing reduction and slicing, the model checkers cannot return any results and UPPAAL cannot return any results in the case of the asynchronous-reactive model either, so values for these are not presented in the tables. Moreover, in the case of *transition-pair* coverage, Theta cannot return results for most properties before running out of memory and there is a great variance in time for the remaining ones, hence the lack of the *DX* rows in Table 10.

6.2.1 Addressing Q-1

The results show that the presented model processing techniques have a significant impact on both the model mapping and model checking time and thus, on the entire test generation process.

Table 9 *Median time* (ms) to map the *railway path locking topology* models into analysis models.

Median transformation time (ms)

M	R&S	Sync	Casc	Async
DU	✗	906	814	1097
	✓	567	541	1304
DX	✗	6437	6304	–
	✓	1000	985	–
XU	✗	6589	6534	–
	✓	1036	1013	–

As for *model mapping*, the effects of the reduction and slicing algorithms can be examined according to whether there are removable elements in the underlying model or not. The algorithms impose a little overhead (a median of 27.2%) when there are no removable elements in the system model as presented in Table 7 (*simple space mission* model). In turn, when the model contains removable elements, these algorithms hardly have any negative effect, the overhead is minimal at worst (a median of 9.9% in the case of *DU* configurations in the *signaller subsystem* and 18.9% in the case of the asynchronous *DU* configuration in the *railway path locking topology*), and in some cases can significantly speed up the transformation process since they decrease the number of elements that need to be mapped as shown in Table 5 (a median of 9.1% gain for *DX* and *XU* configurations) and especially in Table 9 (a median of 84.3% gain for all configurations). It is worth noting that in certain configurations, the gain can be as high as sixfold in this last case.

As for *model checking*, the model reduction and slicing algorithms decrease the necessary generation time in each configuration. Depending on the model, the gain can be significant, ranging

Table 10 *Median time* (s) to generate (top) and the *number of steps* (bottom) in the **transition-pair** coverage test set for the *railway path locking topology* composite model variants.

Median generation time (s)

M	T	Sync	Casc
DU	✗	34.7	15.9
	✓	29.0	14.0
XU	✗	241.4	66.7
	✓	236.6	65.3

#Generated steps

T	Sync	Casc
✗	36	14
✓	20	9
Δ	44%	36%

from 12.6% to 63.9% with a median of 52.5% in the *signaller subsystem* (considering data also for the more coarse-grained interaction criterion in addition to Table 6), and there are also models where model checking is not even possible without such optimization techniques (*railway path locking topology* – Table 10). In general, we can conclude that the effects of the model reduction and slicing algorithms highly depend on the size of the underlying models as well as their characteristics and use in a certain structure, as these are the factors that determine the number of removable elements. Even though, Q-1 focuses on generation time, we also made observations on memory consumption, which strongly correlated with model checking time. Regarding memory consumption in the case of different composition modes, the general results were very similar to that of the case studies in [11].

As for the *abstract test optimization algorithms*, we can also conclude that their employment decrease generation time in each configuration. The greatest median gain can be seen in Table 6 (14.2%) and in the case of *transition* coverage in the *railway path locking topology* (24.9% – data is available in the repository) – in the other cases, the median gains range between 7% and 10%. However, note that these results are median values, and the time-decreasing effect of the first test optimization algorithm highly depends on the order of the properties to be satisfied, which was random in these runs (recall that the decrease in *test size* is deterministic due to the second algorithm). A possible future direction in improving these algorithms could be determining an adequate property order. Furthermore, regarding the concrete models of our case study, we must note that in cases where there is no optimization possibility in the test set, these algorithms will just increase execution time with no gain.

In conclusion, we can state that we have initial evidence that our model processing techniques are gainful according to multiple aspects and their employment can highly facilitate the entire test generation process.

6.2.2 Addressing Q-2

In general, we could expect that test generation time in the case of different composition modes highly depends on the traits of the resulting target (analysis) models. By comparing results for *synchronous-reactive* and *cascade* models, we can conclude that test generation for cascade models compared to synchronous-reactive models is faster in almost every configuration. This is the result of the modeling constructs used to describe concurrent behavior in the analysis models: synchronous-reactive models contain twice as many variables for modeling the supported lockstep-like execution and inter-component communication compared to cascade models. As a result, communication between components in cascade models requires fewer steps (execution turns).

In some cases, the difference in test generation time between model variants is rather little (see the XU results in Table 8 – the median gain for cascade models range between 5% and 10%) and there are cases with a difference as high as fifteenfold (see the XU results in Table 6). Accordingly, if one wants to choose an easy to verify, deterministic composition semantics (which must, of course, conform to the semantics of the implementation) it is best to go with the cascade composition mode.

Model checking of asynchronous behavior is generally more cumbersome than synchronous behavior, which is reflected by the results for the *asynchronous-reactive* composition mode. Test generation without the reduction and slicing algorithms cannot be carried out by UPPAAL in any of the used models as the state space of the unreduced asynchronous-reactive model variants is too large for it to handle. Even when using these reduction techniques, test generation time for *satisfiable* properties is around two orders of magnitude higher than in the case of synchronous-reactive models in the *signaller subsystem* (see Table 6) whereas the difference is around one order of magnitude in the case of the *simple space mission* model (see Table 8). Nevertheless, the test optimization algorithm can mitigate this a little by decreasing test generation time by around 10%. We must also state that the entire state space could not be explored by any of the asynchronous-reactive model variants, which was demonstrated by the inability of UPPAAL to return traces for possibly *uncoverable* elements. These problems could be mitigated by examining and introducing execution constraints regarding the behavior of asynchronous components.

6.2.3 Addressing Q-3

Our model transformations and integrated model checker back-ends support various kinds of analysis models and verification algorithms. The *DU* transformation derives timed automata models that are explored with the explicit-state UPPAAL model checker whereas *DX* and *XU* mappings derive transition system models verified by Theta (using CEGAR-based symbolic techniques) and UPPAAL. UPPAAL also provides efficient algorithms for handling timed behavior. Our expectations were that UPPAAL would perform better at trace generation than Theta due to its explicit state-techniques whereas Theta would have an advantage at proving the unreachability of properties, as similar conclusions were drawn in [30]. We can say our expectations were proven to be true.

The *DU* configuration performs generally well on every system model and composite variant. It provides the best results on the *signaller subsystem* (see Table 6) and the *railway path locking topology* (see Table 10). The difference can be particularly significant compared to other configurations in the case of the synchronous-reactive variants: advantages range from around threefold to even sixfold in some cases. In addition, this is the only configuration supporting the asynchronous-reactive composition mode, although, with somewhat poor results. However, we expect that other model checkers might perform better on such models and will work on this aspect in the future.

The *DX* configuration is always the slowest on every system model (that Theta can handle) if we consider end-to-end test generation, which is generally explicable with the traits of the resulting models and the functioning of the Theta model checker: the transitions of the statechart models have many considerably large guard expressions (monolithic expressions), which are difficult to handle by its current algorithms (more specifically the underlying SMT-solvers). We also know that the algorithms for deriving concrete traces from the abstracted models in Theta are not optimal. Accordingly, there are configurations in the case of every system model that Theta cannot handle. We aim to mitigate these problems by introducing a postprocessing step on the monolithic expressions, disassembling them into smaller parts while introducing control locations to store which parts of the original expressions have been processed. However, based on subresults (not shown in the tables) we noticed that Theta performed reasonably better than UPPAAL at proving the unreachability of properties in some cases, which is not surprising based on the characteristics of the used model checker algorithms. Nevertheless, as can be seen, these gains could not make up for time lost during checking reachable properties.

The *XU* configuration performs rather well in general and has a significant advantage over *DU* on the *simple state mission* model (its advantage is around fivefold – Table 8) and *DX* in every case. In other cases, this configuration has a disadvantage compared to *DU* on synchronous-reactive models and performs similarly or slightly better on cascade models than its *DU* counterpart (see Tables 6 and 10). In general, we can conclude that this *XU* configuration is sensitive to the doubled number of variables introduced for the lockstep-like execution.

In addition to the three selected system models, we carried out measurements on a commercial statechart component called *Flight Guidance System* first presented in [30], which measures the state of an aircraft (position, speed, and altitude) and generates commands to minimize the deviation between the measured and desired state. The model variant we defined in GSL has 43 independent input events controlling 42 transitions and 49 states in 25 orthogonal regions related to specific operational modes that process events *independently* of each other without using any variables. We aimed at full transition coverage with *DU*, *DX* and *XU* models. UPPAAL could not handle the large state space caused by the great number of input events: an out of memory error was thrown after around 30 seconds both on *DU* and *XU* models. In turn, Theta could generate the entire transition covering test set spending a few seconds on each property, proving the utility of symbolic techniques in the case of such abstractable models and the relevance of Theta in our tool.

To conclude, the results show the versatility the various analysis models and integrated model checkers add to the test generation approach. The different target models of UPPAAL function well on models with moderate size and are generally good for returning traces for reachable properties. The results of Theta are poorer in general due to its difficulties when returning

concrete traces; however it has generally good results for abstractable models and proving the unreachability of properties, and thus, can be a good choice when we expect many unreachable properties, serving as a good complementation to UPPAAL.

6.3 Conclusion and threats to validity

In conclusion, the results highlight the merits of the multiple supported optimization algorithms, integrated model checker back-ends and analysis models. The reduction, slicing and test optimization algorithms can significantly speed up the generation process with little to no cost, and the different supported formalisms and model checker back-ends excel at handling different model types, test criteria and property sets, bringing a large amount of flexibility into the approach and Gamma in general. The tool performs well in the case of synchronous-reactive and cascade model variants with coarse-grained and finer test criteria, too; however, performance issues emerge in the case of the asynchronous-reactive variant. These issues could be addressed by introducing pragmatic constraints and support for asynchronous models in the *DX* and *XU* transformations and integrating algorithms tailored to asynchronous behavior into Theta (see Sect. 8).

Subject to threat to validity, we can highlight the following aspects. First, we constructed the *simple space mission* model manually based on semi-formal graphical descriptions, while taking into account the characteristics of the Gamma languages. This approach resulted in a model tailored to the modeling languages of the framework, potentially distorting the results compared to fully automatic mappings. Next, we evaluated our tool in the context of a limited set of (three) system models. Also, the resulting test sets could be optimized in every model as there were always traces that were prefixes to others. Thus, our optimization algorithms in most cases could significantly speed up the generation process, which could not happen with unoptimizable test sets. Moreover, the effects of the first test optimization algorithm highly depend on the order of the verifiable properties and thus, executions with different property orderings can substantially differ. Finally, the tool generates tests using a series of complex automated model transformations in Gamma: potential implementation flaws in the transformations can result in semantically incorrect analysis models, distorting the results. Nevertheless, all generated tests passed, demonstrating the semantic equivalence of the derived analysis models and implementation for the *examined traces* (including time-dependent behavior) and showing the good quality of the model transformations in general.

7 Related work

This section presents MBT approaches and tools that target *(i)* state-based models and build on *(ii)* model checkers and test coverage criteria as these are the key aspects of the testable systems and our test generation approach in Gamma. Accordingly, we exclude approaches that target different model types, test selection criteria and technologies [32–34] as well as solutions where test generation characteristics are not presented in detail. Although our solution targets integration testing, we also include works focusing on standalone reactive components (i.e., single statechart models). We organize related work in accordance with the supported coverage criteria, that is, we present solutions for *logical*, *model element-based* and *behavior-based* coverage criteria.

7.1 Solutions for logical condition coverage criteria

Authors in [35] present a method for the automatic generation of test cases to cover structural or logical coverage criteria (e.g., modified condition and decision – MC/DC) for standalone transition systems. The solution supports analyzing SCR and RSML requirement specifications or Java source code (front-end languages). Similarly to ours, the method uses multiple model checkers (Spin and SMV) and automatically generated LTL properties to cover paths in the models.

In [36], a tool-supported approach (CompleteTest) is presented to analyze software written in the Function Block Diagram (FBD) language. The approach supports logical coverage criteria (e.g., MC/DC) while mapping FBDs into the timed automata formalism of UPPAAL to generate tests automatically. The results show that the

approach scales well for the examined FBDs and is applicable in industrial practice.

In [37], the AutoMOTGen tool suite is presented, which supports the automatic mapping of Simulink/Stateflow models into the SAL (Symbolic Analysis Laboratory) framework and the model checking based generation of test cases based on various logical coverage criteria (e.g., MC/DC). The authors give a comparative study with a random input-based test generation tool and conclude that model checking-based techniques can complement the random techniques.

7.2 Solutions for model element-based criteria

AGEDIS [38] is an MBT tool suite for component-based distributed systems integrating model and test suite editors, test case simulation and debugging tools, a test coverage analysis and defect analysis tools as well as report generators. Test models can be defined in the form of UML class diagrams, state machine diagrams and object diagrams (to specify the SUT initial state). The test generator builds on the TGV engine [39] and supports state and transition coverage.

Smartesting CertifyIt [40] is also an MBT tool suite integrating editors for requirements definition and traceability, test adapter and test models. Test models comprise UML class diagrams (to describe data), state machines and object diagrams (initial states of executions) and BPMN notations. Tests can be generated using the CertifyIt Model Checker, supporting state, transition and transition-pair coverage.

Authors in [41] present an approach for combining explicit-state, symbolic-state and bounded model checkers in the SAL framework by building on results presented in [30] to generate efficient test sets based on standalone Stateflow models. Tests are efficient in terms of both generation and execution, and are generated by iterated extensions of previously retrieved paths. The authors compare the usability of these model checking techniques and propose efficient algorithms for combining the strengths of different model checkers for state and transition coverage in state-based models; a promising technique that our approach could also benefit from in the future.

In [42], authors focus on generating test cases from standalone statechart models to achieve state and transition coverage by translating them into the SMV language and using the NuSMV model checker. They also propose a test optimization technique similar to ours based on finding prefixes.

In [43], authors focus on deriving test cases from standalone statecharts to achieve various coverage criteria. Similarly to our approach, they introduce control-flow (e.g., MC/DC, state and transition) and dataflow coverage criteria and adapt them to statecharts. They map statecharts and coverage criteria into the input language of the SMV model checker.

7.3 Solutions for behavior-based criteria

In [44], the Component Interaction Testing project is introduced supporting an object-oriented modeling language (ObjectState) with formal semantics based on the LTS formalism to define interaction coverage criteria between state-based models. Algorithms reducing state space explosion for test generation are also presented and rely on the iterative expansion of partial test cases and the abstraction of interactions between components (while environmental interactions are kept).

Authors in [45] aim at the testing of service choreographies using model-based integration testing techniques. The Message Choreography Models (MCM), which describe communication protocols between services, are transformed into Event-B models and used as input for their in-house ProB model checker to generate, e.g., event- and transition-covering test suites. They also propose a test reduction algorithm based on every possible combination of test cases that selects the optimal one according to their specified objectives.

7.4 Novel capabilities and evaluation aspects in our work

Our approach focuses on the generation of integration tests for reactive components utilizing the multiple composition semantics and model checkers of the Gamma framework. In the literature, several MBT approaches were proposed for model checker based test generation; many of these use multiple back-ends with various techniques (explicit-state, symbolic, bounded) – a solution

our approach also relies on. Some test optimization techniques were also presented in terms of generation time and size similar to ours.

Nevertheless, we did not find any research in the literature focusing on the (mix-and-match) integration of components according to *different composition semantics*: the examined approaches either target standalone components or a *single composition mode* supporting only a single front-end language. Moreover, they do not consider the effects of possible *model preprocessings* (reductions) and *test target configurations*.

In our work, we aimed to make contributions in these aspects. In our evaluation, we examined various systems with *different traits* integrated according to *different composition semantics* and investigated the effects in terms of test generation time and the size of generated test sets. We also examined the effects of our *model reduction* techniques and *test target configuration* capabilities.

8 Conclusion and future work

This paper introduced an MBT approach in the Gamma Statechart Composition Framework for heterogeneous state-based components integrated according to various execution and interaction semantics. The test generator tool provides a multitude of configurable and extensible structural, dataflow- and behavior-based coverage criteria for integration testing based on model queries and utilizes integrated model checker back-ends for test generation. Results show that the approach and tool are applicable on synchronous models from industrial practice. The supported analysis models of different model checkers are suited to different behaviors, complementing each other and the model reduction, model slicing and test optimization algorithms provide significant gains during the process with close to zero cost. Nevertheless, the verification of asynchronous models poses a great challenge for the tool, which could be mitigated by introducing pragmatic restrictions on their behavior.

For future work, we plan to change the query language of GPPL to that of VIATRA to increase its expressive power. We also plan to improve the verification of asynchronous models based on the approach presented in [46] and introduce techniques tailored to such behavior into the Theta model checker back-end, e.g., saturation [47]. In addition, we aim to integrate SysMLv2 [48] into the framework to aid industrial parties in the semantically sound composition and analysis of system models.

Acknowledgement This work was partially supported by the ÚNKP-20-3 New National Excellence Program of the Ministry for Innovation and Technology.

Data availability All data generated or analyzed during this study are included in this article and the public repository https://github.com/ftsrg/gamma.

Conflict of interest All authors declare that they have no conflicts of interest.

References

[1] George T Heineman and William T Councill. Component-based software engineering. *Putting the Pieces Together, Addison Wesley*, 2001.

[2] 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 Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing*, pages 235–248. Springer, 2014.

[3] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, May 2011.

[4] 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):427–451, 2016.

[5] Adam Childs, Jesse Greenwald, Georg Jung, Matthew Hoosier, and John Hatcliff. CALM and Cadena: Metamodeling for component-based product-line development. *IEEE Computer*, 39(2):42–50, 2006.

[6] 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)*, pages 199–208, Aug 2007.

[7] 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, editors, *Tools and Methods of Program Analysis*, pages 77–89, Cham, 2018. Springer International Publishing.

[8] Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for test case generation. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, pages 391–438. Springer, 2005.

[9] 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*, pages 1–6, 2010.

[10] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma Statechart Composition Framework. In *40th International Conference on Software Engineering (ICSE)*, pages 113–116, Gothenburg, Sweden, 2018. ACM.

[11] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling*, 19(6):1483–1517, 2020.

[12] Ármin Zavada. Formal modeling and verification of process models in component-based reactive systems. Technical report, Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems, 2021.

[13] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.

[14] Gianfranco Ciardo and Radu Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 40–53. Springer, 2003.

[15] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000*, pages 103–122. Springer, 2000.

[16] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Introduction to Model Checking*, pages 1–26. Springer, Cham, 2018.

[17] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, QEST '06, page 125–126, USA, 2006. IEEE Computer Society.

[18] 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, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017.

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

[20] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Elsevier, 01 2007.

[21] Bence Graics. Documentation of the Gamma Statechart Composition Framework v2.0. Technical report, Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems, 2018.

https://tinyurl.com/2xxyujtf.

[22] 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):020125, 2017.

[23] Sandra Rapps and Elaine Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367– 375, 05 1985.

[24] 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):609–629, 2016.

[25] Bence Graics. Documentation of the Gamma Statechart Composition Framework v0.9. Technical report, Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems, 2016. https://tinyurl.com/yeywrkd6.

[26] Milán Mondok. Extended symbolic transition systems: an intermediate language for the formal verification of engineering models. Technical report, Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems, 2020. http://tdk.bme.hu/VIK/DownloadPaper/Kiterjesztett-szimbolikus-tranzicios.

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

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

[29] 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)*, pages 23–32, 2005.

[30] 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, editors, *Formal Approaches to Software Testing*, pages 42–59. Springer, 2004.

[31] M.P.E. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria. In *Eighth IEEE International Symposium on High Assurance Systems Engineering, Proceedings.*, pages 178–186, 2004.

[32] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

[33] Muhammad Shafique and Yvan Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.

[34] Havva Gulay Gurbuz and Bedir Tekinerdogan. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal*, 26(4):1327–1372, 2018.

[35] 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*, pages 83–91. IEEE, 2001.

[36] 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):335–353, 2016.

[37] 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.*, 24:155–180, 2014.

[38] Alan Hartman and Kenneth Nagin. The AGEDIS tools for model based testing. *ACM Sigsoft Software Engineering Notes*, 29, 07 2004.

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

[40] 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*, pages 391–397, 2013.

[41] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM)*, pages 261–270, 01 2004.

[42] 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*, pages 37–42, 2009.

[43] Hyoung Hong, Insup Lee, and Oleg Sokolsky. Automatic test generation from statecharts using model checking. *Technical Reports (No. MS-CIS-01-07)*, 10 2001.

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

[45] 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, editors, *Testing of Software and Communication Systems*, pages 179–194. Springer, 2009.

[46] 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*, MODELS, pages 1–5. ACM, 2020.

[47] 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*, pages 342–361. Springer, 2019.

[48] Systems Modeling Language Version 2 (SysMLv2). Standard, Object Management Group (OMG), December 2020.

Appendix

The Appendix contains additional information about the modeling languages of the Gamma framework, the annotation process of composite models during test generation and composite system models on which the experimental evaluation of the test generation approach was carried out.

A1 Gamma modeling languages

This section details the Gamma Statechart Language, Gamma Composition Language and Gamma Trace Language.

A1.1 Gamma Statechart Language

The Gamma Statechart Language (GSL) serves as a common representation language for component statecharts and supports different statechart semantics by means of annotations. GSL, like every Gamma language, organizes models into packages and supports their import using a relative path, e.g., interfaces, serving as realizable communicational contracts. In GSL models, annotations can be used to specify conflict resolution

between transitions (*@RegionSchedule*) and priority between transitions with the same source (*@TransitionPriority*) to represent the behavior of the original Yakindu models. In addition, ports are defined as interaction points via which the component can communicate with its environment. Finally, variable declarations (with potential annotations, detailed in Sect. 2.3), regions with state nodes and transitions are defined.

A1.2 Gamma Composition Language

In the Gamma Composition Language (GCL), the definition of composite models in different composition modes differ only in a single keyword (*sync*, *cascade* or *async*); the definition of *ports* (they are identical to statechart ports), contained *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.

```
package cabincontrol
import "Interface/Interface.gcd" // Importing interfaces
     for ports (relative path)
@RegionSchedule = bottom-up // Conflict resolution between
      hierarchy levels
@TransitionPriority = order-based // Priority between
     transitions with the same source
statechart CabinControl [
 // Ports via which events can be sent and received
 port Door : provides Door
 port Cabin : requires Cabin
] {
 // Variable declarations
 var isDoorClosed : boolean
 // Regions and contained state nodes
 region main {
  initial Entry
  state TravellingDown
  state Idle
  state TravellingUp
  choice Choice1
  choice Choice2
 }
 // Transitions
 transition from Entry to Idle
 transition from Idle to Idle when Cabin.close /
  raise Door.close;isDoorClosed := true;
 transition from Idle to Idle when Cabin.open /
  raise Door.open; isDoorClosed := false;
 transition from Idle to Choice1 when Cabin.up_
 transition from Idle to Choice2 when Cabin.down
 transition from TravellingDown to Idle when Cabin.stop
 transition from TravellingUp to Idle when Cabin.stop
 transition from Choice1 to TravellingUp [isDoorClosed]
 transition from Choice1 to TravellingUp [else] /
  raise Door.close; isDoorClosed := true;
 transition from Choice2 to TravellingDown [isDoorClosed]
 transition from Choice2 to TravellingDown [else] /
  raise Door.close; isDoorClosed := true;
}
```

Fig. 16 GSL *cabin controller* of the *elevator system*.

```
... // Package and annotation definitions as well as
     interface import
statechart CabinDoorControl [
 port Door : requires Door
] {
 region main {
  initial Entry
  state Open
  state Closed
 }
 transition from Entry to Closed
 @(main_closed__main_open) // Id annotation
 transition from Closed to Open when Door.open
 @(main_open__main_closed) // Id annotation
 transition from Open to Closed when Door.close
}
```

Fig. 17 GSL *cabin door controller* of the *elevator system*.

```
package elevator
import "Interface/Interface.gcd"
// Importing the Gamma statechart models
import "CabinControl/CabinControl.gcd"
import "DoorControl/CabinDoorControl.gcd"
// Synchronous-reactive, cascade or asynchronous-reactive
     model according to the keyword
[sync / cascade / async] Elevator [
 port Cabin : requires Cabin
] {
 // Component instances of the system
 component cabinControl : CabinControl
 component cabinDoorControl : CabinDoorControl
 // Binding system ports to ports of contained components
 bind Cabin -> cabinControl.Cabin
 // Channels for component communication
 channel [cabinControl.Door] -o)- [cabinDoorControl.Door]
}
```

Fig. 18 GCL synchronous-reactive, cascade or asynchronous-reactive composite model of the *elevator system*.

In our elevator example, we define a composite model (*Elevator*) with a single port named *Cabin*. We also define two component instances *cabinControl* and *cabinDoorControl* of GSL statechart types *CabinControl* and *CabinDoorControl* that define the behavior of the composite model. The single system port is bound to port *Cabin* of the *cabinControl* component, that is, every event received via this system port is processed by *cabinControl*. Finally, the defined *channel* enables event transmission between port *Door* of the *cabinControl* component and port *Door* of the *cabinDoorControl* component.

A1.3 Gamma Trace Language

The Gamma Trace Language is a high-level trace language tailored to the characteristics of GCL

```
// E F (var main_open__main_closed)
trace OpenClosedTrace of Elevator
step {
 act {
  reset
 }
 assert {
  cabinControl.Idle
  cabinDoorControl.Closed
  !cabinControl.isDoorClosed
  !cabinDoorControl.main_closed__main_open
  !cabinDoorControl.main_open__main_closed
 }
}
step {
 act {
  raise Cabin.open
  schedule component
 }
 assert {
  cabinControl.Idle
  cabinDoorControl.Open
  !cabinControl.isDoorClosed
  cabinDoorControl.main_closed__main_open
  !cabinDoorControl.main_open__main_closed
 }
}
step {
 act {
  raise Cabin.down
  schedule component
 }
 assert {
  cabinControl.TravellingDown
  cabinDoorControl.Closed
  cabinControl.isDoorClosed
  !cabinDoorControl.main_closed__main_open
  cabinDoorControl.main_open__main_closed
 }
}
```

semantics, supporting the description of execution traces for composite models. Such execution traces formally describe the behavior of composite models, that is, what states it goes into (state configuration and variable values) and what events it produces (*asserts* in short) in response to certain inputs (*acts*) from the environment (input events, time lapse and scheduling). GTL also supports the specification of undesired behavior or the combination of different behaviors, i.e., the combination (And, Or, Xor-relation) of undesired outputs or variable values in specific states.

A2 Annotating composite models

The annotation of composite models (ignoring excluded elements) takes place in accordance with the selected coverage criteria as follows.

State and out-event coverage

In the case of *state* and *out-event* coverage, no auxiliary elements are inserted into the model.

Transition coverage

In the case of *transition* coverage, a boolean variable (marked *resettable*) is created for every transition with a *false* initial value. Next, the action list of every transition is extended with an assignment action that sets the corresponding variable to true. This way, the variable is true only if the corresponding transition fires.

Transition-pair coverage

In the case of *transition-pair* coverage,

- a pair of integer variables (*previousId* and *actualId*) is created for every state that has both incoming and outgoing transitions;
- every transition entering or leaving such a state is assigned a unique integer identifier;
- the action list of every incoming transition is extended with an action that saves the identifier of the transition in the *actualId* variable;
- the *exit* action list of each state is extended with an action saving the value of the *actualId* variable in the *previousId* variable;

```
// E F (var main_closed__main_open)
trace ClosedOpenTrace of Elevator
step {
 act {
  reset
 }
 assert {
  cabinControl.Idle
  cabinDoorControl.Closed
  !cabinControl.isDoorClosed
  !cabinDoorControl.main_closed__main_open
  !cabinDoorControl.main_open__main_closed
 }
}
step {
 act {
  raise Cabin.open
  schedule component
 }
 assert {
  cabinControl.Idle
  cabinDoorControl.Open
  !cabinControl.isDoorClosed
  cabinDoorControl.main_closed__main_open
  !cabinDoorControl.main_open__main_closed
 }
}
// Note that it is prefix of OpenClosedTrace
```

Fig. 19 GTL execution traces derived for the transition coverage properties of the *cabin door controller* component of the *elevator system*.

```
public class ClosedOpenTrace {
 // Generated implementation wrapped in a reflective API
 private static ReflectiveElevator elevator;
 @Before
 public void init() {
  elevator = new ReflectiveElevator();
 }
 public void step0() {
  // Initializing the implementation
  elevator.reset();
  // Assert
  assertTrue(elevator.getComponent("cabinControl")
   .isStateActive("main", "Idle"));
  assertTrue(elevator.getComponent("cabinDoorControl")
   .isStateActive("main", "Closed"));
  assertTrue(elevator.getComponent("cabinControl")
   .checkValue("isDoorClosed", false));
 }
 @Test
 public void finalStep() {
  step0();
  // Raising Cabin.open with no argument
  elevator.raiseEvent("Cabin", "open", new Object[] {});
  // Initiating an execution cycle
  elevator.schedule();
  // Assert
  assertTrue(elevator.getComponent("cabinControl")
   .isStateActive("main", "Idle"));
  assertTrue(elevator.getComponent("cabinDoorControl")
   .isStateActive("main","Open"));
  assertTrue(elevator.getComponent("cabinControl")
   .checkValue("isDoorClosed", false));
 }
}
```

Fig. 20 JUnit test derived for the transition-covering test set of *cabin door controller* of the *elevator system*.

- the action list of every outgoing transition is extended with an action that saves the identifier of the transition in the *actualId* variable.

Consequently, the values of the *previousId* and *actualId* variables for a certain state are n and m, only if the last activation and deactivation of the state were executed by the incoming transition with identifier n and the outgoing transition with identifier m. Note that this approach supports the correct handling of loop transitions.

Interaction coverage

In the case of *interaction* coverage,

- every event raise action (called *sender* in general), that is, entry or exit action of a state or action of a transition (this latter is called *transition sender* in general), is assigned an integer identifier according to the specified *sender-coverage-criterion*:
  - *events* – senders interacting via the same port-event combination in a component are assigned the same identifier; in all other cases, senders are assigned different identifiers;
  - *states-and-events* – transition senders that *1)* interact via the same port-event combination and *2)* are contained by transitions leaving the same state in a component, are assigned the same identifier; in all other cases, senders are assigned different identifiers;
  - *every-interaction* – every sender is assigned a unique identifier.
- every transition with a trigger (called *receiver* in general) is assigned an integer identifier according to the specified *receiver-coverage-criterion*:
  - *events* – receivers triggered by the same port-event combination in a component are assigned the same identifier; in all other cases, receivers are assigned different identifiers;
  - *states-and-events* – receivers that *1)* are triggered by the same port-event combination, and *2)* leave the same state in a component, are assigned the same identifier; in all other cases, senders are assigned different identifiers;
  - *every-interaction* – every receiver is assigned a unique identifier.
- the parameter list of every event declaration raised by a sender is extended with an integer parameter named *senderId* that will be responsible for storing the identifier of the sender;
- the argument list of every sender is extended with its identifier;
- in every atomic component containing receivers, a pair of integer variables, named *senderId* and *receiverId* (both marked *resettable*) are created for each region that will be responsible for storing the identifier of the sender and receiver of the last interaction in a particular region;
- the action list of every receiver transition is extended with an action, saving the identifier of the sender and receiver in the corresponding variables.

Accordingly, the values of the *senderId* and *receiverId* variables for a certain region are n and m, only if the last interaction in that particular region took place between a sender of another component with identifier n and a receiver transition of that certain region with identifier m. Note that in a region, only one transition can fire in a single execution turn and thus, these

variables cannot be overwritten. As an example, Fig. 21 presents the annotated statechart models of the *elevator system* during interaction coverage after setting both receiver-coverage-criterion and sender-coverage-criterion to *every-interaction*.

Dataflow coverage

In the case of *internal dataflow* coverage, an integer variable is created for every considered variable declaration (denoted by d). Furthermore, a unique integer identifier is assigned to every variable definition statement (*def* d), and an integer variable (marked *resettable*) is created for every declaration use (*use* d). An assignment statement is inserted after each *declaration definition* statement, saving the identifier of *def* d in d. Moreover, an assignment statement is inserted into the corresponding action list after each *declaration use*, saving the value of the corresponding d variable in the corresponding *use* d variable (see an example in Fig. 22). The difference in the case of *interaction dataflow* coverage is that instead of integer variable declarations, parameter declarations of type integer (denoted by d) are created for event declarations containing considered parameter declarations, and unique integer identifiers are assigned to every event raise action (*def* d); their argument list also gets extended with their identifier (similarly to the approach presented in the case of *interaction* coverage). This approach facilitates the identification of *def-clear paths* from different declaration definition statements: in the case of a d declaration, the integer variable of a *use* d statement is n only if a *def-clear* d *path* from *def* d with identifier n to *use* d is covered. Note that this annotation method is independent of the selected coverage criterion; the coverage criterion affects only the generation of properties (see Sect. 4.3).

```
statechart CabinControl [...] {
 ...
 // Transitions with with raise event actions sending
      their identifiers as arguments
 transition from Choice1 to TravellingUp [else] /
  raise Door.close(1); isDoorClosed:= true;
 transition from Choice2 to TravellingDown [else] /
  raise Door.close(2); isDoorClosed := true;
 transition from Idle to Idle when Cabin.close /
  raise Door.close(3); isDoorClosed := true;
 transition from Idle to Idle when Cabin.open /
  raise Door.open(4); isDoorClosed := false;
}
statechart CabinDoorControl [...] {
 // Resettable variables for storing the ids of
      interacting senders and receivers
 @Resettable var senderId : integer
 @Resettable var receiverId : integer
 ...
 // Transitions extended with actions saving the ids of
      the sender and receiver
 transition from Closed to Open when Door.open /
  senderId := Door.open::senderId; receiverId := 5;
 transition from Open to Closed when Door.close /
  senderId := Door.open::senderId; receiverId := 6;
}
```

Fig. 21 Annotated *cabin controller* and *cabin door controller* models of the *elevator system*: both receiver-coverage-criterion and sender-coverage-criterion are set to *every-interaction*.

A3 System models for evaluation

This section presents the system models on which the evaluation of our test generation approach is carried out.

A3.1 Signaller subsystem for railway traffic control

The *signaller subsystem* consists of three atomic statechart components: two identical *antivalence checkers* and one *signaller*. The antivalence checkers (defined in MagicDraw, depicted in Fig. 10) sample input signals (h and l, standing for *high* and *low*) coming from the environment, potentially handle inconsistencies and transfer the signals (t, f and p, standing for *true*, *false* and *previous*, respectively) to the connected signaller. The handling of inconsistency in input signals can be set by a parameter (*P_STORE*), which turns error storage on or off – we set this parameter to true (this will eventuate a greater state space during verification). The signaller (defined in Yakindu, depicted in Fig. 11) sends signals to the environment (*ACTIVE* and *PASSIVE* signals sent via the *O* port) according to a well-defined railway control protocol based on inputs coming from the antivalence checkers via the *I_FT* and *I_CR* ports. Both components rely on an external timer component, responsible for sending timeout signals after specified time intervals (*timeout* triggers in the models).

A3.2 Simple space mission

The *simple space mission* system model comprises two communicating statechart components among which data transmission takes place: a *ground station* (see Fig. 12) and a *spacecraft* (see Fig. 13).

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 transmitting data upon the reception of a *ping* event 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.

As the original model was simulated using the Cameo Simulation Toolkit[16] with discrete time steps, we can model the system with the synchronous-reactive composition mode. In addition, we create a cascade and an asynchronous-reactive variant to analyze the effects of different interaction and communications semantics.

As an example, Fig. 22 shows and excerpt of the annotated *spacecraft* model to support *all-def*, *all-c-use/all-p-use* and *all-use* dataflow coverage criteria according to the rules presented in Section A2.

```
statechart Spacecraft [
 port connection : provides DataSource
] {
 // Original variables
 var battery : integer := 100
 var recharging : boolean := false
 var data : integer := 100
 // Injected variables for declarations (d)
 var def_battery : integer
 var def_recharging : integer
 var def_data : integer
 // Injected variables for decl. uses (use-d)
 @Resettable var use_battery_0 : integer
 ...
 @Resettable var use_battery_8 : integer
 @Resettable var use_recharging_0 : integer
 @Resettable var use_data_0 : integer
 // Timoeuts, regions and states
 ...
 transition from WaitingPing to Transmitting
  when connection.ping [recharging = false] /
  use_recharging_0 := def_recharging;
 transition from Sending to Sending when
  timeout transmitTimeout [data > 1 and battery >= 40] /
  use_data_0 := def_data; // p-use
  use_battery_0 := def_battery; // p-use
  data := data - 1; // Original def
  use_data_1 := def_data; // c-use
  def_data := 0; // After the use setting:
  // assigning an id (0) to the data def
  raise connection.data;
 transition from Consuming to Consuming when
  timeout consumeTimeout [battery >= 40] /
  use_battery_1 := def_battery; // p-use
  battery := battery - 1; // Original def
  use_battery_2 := def_battery; // c-use
  def_battery := 0; // After the use setting:
  // assigning an id (0) to the battery def
 ...
}
```

Fig. 22 Excerpt of the annotated *spacecraft* model of the *simple space mission* model for dataflow coverage criteria.

A3.3 Railway path locking topology

The *railway path locking topology* consists of two kinds of component models, a *signaller* (a model different from the one presented in the *signaller subsystem*) and a *turnout* (see Fig. 14). Both kinds of components have four ports. Communication with the external control environment can be carried out via port T of the signaller whereas the D, S, E, L and R ports are used for communication among the components of the topology. Via the B ports, the components can receive data from lower-level objects (hardware-related elements). The operation of every port is independent of the others' and thus, communication is defined using orthogonal regions in both model types. Altogether, the signaller model consists of five orthogonal regions, 12 states, 41 transitions and ten variables and has a parameter specifying the role of the instantiated object (*reverse*, *main* or *reverse & main*). The turnout model consists of seven orthogonal regions, 23 states, 81 transitions and seven variables and has a parameter specifying its initial control setting (*left* or *right*, that is, the L or the B port is "connected" to E internally).

The *railway path locking topology* (depicted in Fig. 15) consists of a main signaller, a reverse signaller and two *reverse & main* signallers with turnouts in between them controlled in an adequate state. The main signaller is set to be able to communicate via its T port and initiate a railway path locking until the rightmost signaller upon an external *railway path locking* request and return with the (positive or negative) response. Furthermore, every component can communicate via its B port in addition to the D, S and E, L and R ports connected to neighboring components. Similarly to the *signaller subsystem*, the designers

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

aimed to investigate the effects of different composition semantics in terms of testing and thus, we construct a synchronous-reactive, cascade and an asynchronous-reactive composite model variant.