Model-Based Testing of Asynchronously Communicating Distributed Controllers

Bence Graics[1], Milán Mondok[1], Vince Molnár[1], and István Majzik[1]

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
Műegyetem rkp. 3., H-1111, Budapest, Hungary
{graics,molnarv,majzik}@mit.bme.hu

Abstract. Programmable controllers are gaining prevalence even in distributed safety-critical infrastructures, e.g., in the railway and aerospace industries. Such systems are generally integrated using multiple loosely-coupled reactive components and must satisfy various critical requirements. Thus, their systematic testing is an essential task, which can be encumbered by their distributed characteristics. This paper presents a model-based integration test generation approach leveraging hidden formal methods based on the collaborating statechart models of the components. Statecharts can be integrated using various composition modes (e.g., synchronous and asynchronous) and then transformed (via a symbolic transition systems formalism – XSTS) into the input formalisms of model checker back-ends, namely UPPAAL, Theta and Spin in an automated way. The model checkers are utilized for test generation based on multiple coverage criteria. The approach is implemented in our open source Gamma Statechart Composition Framework and evaluated on industrial-scale distributed controller subsystems from the railway industry.

Keywords: Model-based integration testing · Collaborating statecharts · Asynchronous communication · Hidden formal methods · Tool suite

1 Introduction

Software-intensive programmable controllers are becoming increasingly widespread in safety-critical infrastructure, including railway interlocking systems (RIS) [25,32] and onboard computers of satellites [29,41,1]. Such systems are commonly integrated using multiple components that may communicate in different ways, e.g., synchronously or asynchronously. This way, these components form a loosely coupled distributed architecture. In general, these systems are embedded into their environments and system components must cooperate to conduct complex tasks in response to external commands and environmental or internal events (e.g., changes in the context or component failures) to reach their objectives. Thus, they are often referred to as *reactive systems*.

The distributed architecture of these systems may complicate their development, necessitating precise means to describe the functional behavior of the

components, as well as their integration, including their execution (e.g., sequential, concurrent or parallel) and communication modes (e.g., signal- or message-based). In addition, as these systems carry out safety-critical tasks, testing their implementation in a systematic way is vital. Nonetheless, the verification of component interactions is often encumbered by the imprecise, informal definition and interpretation of their execution and communication modes.

To aid the development of complex distributed systems, model-based and component-based systems engineering (MBSE and CBSE) [21,27,3,39] approaches advocate the application of reusable models and components based on high-level modeling languages, e.g., SysML and UML. These modeling languages allow for the platform-independent description of functional behavior in terms of components, e.g., statecharts [19] for reactive behavior, as well as structure, e.g., block diagrams to define the (hierarchical) integration of system components.

In turn, most MBSE and CBSE approaches do not provide refined and extensible tool-centric means for the automated and systematic testing of the system implementation [30]. These insufficiencies may stem from informal model semantics or the lack of sound and efficient verification methods, e.g., the lack of integration between the high-level models and verification back-ends [9,31].

Consequently, verification-oriented MBSE and CBSE approaches for developing distributed control systems should support – in an integrated tool suite – i) high-level modeling languages with *precise semantics* to describe the behavior of standalone components, and *component integration* (composition) based on different modes of execution and communication, ii) automated *model transformations* with *traceability* that map these models into different *model checkers* to support the *exhaustive verification* of functional behavior in a flexible and reusable way, and iii) *test generation* algorithms suitable for large-scale systems based on various *coverage criteria* to test the system implementation efficiently.

To address these challenges, we propose a fully automated model-based testing (MBT) approach in our open source Gamma Statechart Composition Framework[1] [34]. At its core, the approach builds on a high-level *statechart language* (GSL) [12] and a *composition language* (GCL) [17] with precise semantics to describe the functional behavior of standalone components, as well as their integration using various execution and communication modes (jointly, referred to as *composition* modes) to support the modeling of synchronous and asynchronous systems. The emergent models are automatically mapped into a low-level formalism, called *EXtended Symbolic Transition Systems* (XSTS), serving as a *common formal representation* to capture reactive behavior. *Formal verification* based on temporal properties is supported by mapping XSTS models and properties into the input formalisms of different model checker back-ends; so far, UPPAAL [4] and Theta [40] have been supported. The mappings feature *model reduction* and *model slicing* algorithms to allow for the verification of large-scale systems. Back-annotation facilities automatically map the verification results into a high-level *trace language* (GTL) [17]. Building on the formal verifica-

[1] More information about the framework (e.g., preprints) and the source code can be found at `http://gamma.inf.mit.bme.hu/` and `https://github.com/ftsrg/gamma/`.

tion and back-annotation functionalities [10], the approach generates *integration tests* based on customizable structural (model element-based), dataflow-based and behavior-based (interactional) coverage criteria. Test cases are *optimized* and concretized to different platforms, e.g., C or Java, to detect faults in the implementation of components (e.g., missing implementation of states or transitions), incorrect variable definitions and uses or component interactions.

Our previous works [34,17,16] focused on the design and verification of *timed* systems (with UPPAAL) and reactive systems featuring *signal-* and *shared variable* based communication (with Theta). The already presented version of the framework featured GSL and GCL, the model transformations into the UPPAAL and Theta model checker back-ends along with the model reduction and slicing algorithms, and automated test generation based on the coverage criteria. This work extends the framework to support *asynchronous* systems with *message-based* communication and integrates the open source Spin [23,24] model checker, which is tailored to verifying models with such characteristics. The integration necessitates i) a mapping between the framework's behavioral representation (XSTS) and Promela (the input language of Spin) and ii) the validation of our approach, including the evaluation of its performance. The results show that Spin provides additional versatility and thus, is a valuable addition to the framework.

The novel contributions of the paper are as follows:

1. the *EXtended Symbolic Transition Systems* (XSTS) formalism (see Sect. 4) as a *general representation* for reactive components and their integration that supports high-level control structures (see their application in Sect. 5) to facilitate efficient verification;
2. a nontrivial *mapping* between the XSTS and Promela languages (see Sect. 6) integrating the open source Spin model checker into Gamma, including two message queue representation modes for asynchronous communication using i) *arrays* and ii) native *asynchronous channels* in Promela; and
3. the *evaluation* of our extended approach on real-life distributed controller subsystems from the railway industry, comparing the UPPAAL, Theta and Spin integrated model checkers and their underlying algorithms (see Sect. 7).

2 Related Work

The main features of our approach revolve around i) a low-level analysis language (XSTS) to describe reactive behavior and allow for formal verification using model checkers and ii) an end-to-end integrated MBT approach for composite high-level engineering models using hidden formal methods (i.e., users do not have to familiarize themselves with the underlying analysis models and algorithms). Thus, we present related work according to these aspects.

Analysis languages BoogiePL [6], as the input formalism of the Boogie model checker, is a generic intermediate language for verifying object-oriented programs. The language is coarsely typed and offers constructs such as procedures and arrays. The main distinguishing feature of BoogiePL compared to XSTS is

that it was specifically tailored to fit *computer programs*, not reactive behavior, and offers constructs for this specific domain (like procedures).

PVS [36] is a specification and verification language based on classical, typed higher-order logic and can be used for *theorem proving*, a lower-level approach to construct and maintain large formalizations and logical proofs.

The Symbolic Analysis Laboratory (SAL) [38] language is an intermediate language tailored to *concurrent transition systems*. It is intended to be the target for translators that extract the behavior descriptions from other languages, and a common source for different analysis tools. It is supported by the SAL toolset that includes symbolic (BDD-based) and bounded (SAT/SMT-based) model checkers. In turn, SAL provides limited support for traceability and back-annotation with respect to the integration of high-level modeling languages.

In comparison, we offer XSTS, an extension of *symbolic transition systems* close to the input formalisms of model checkers, which also includes *control structures* to retain several characteristics of the engineering models that can be utilized in mappings for model checking (e.g., parallel execution), and also supports easy traceability and back-annotation regarding the verification results.

Integrated test generation approaches The idea of using *hidden formal methods* to *generate tests* based on *integrated models* has been applied in several tool-based approaches. The CompleteTest tool [8] analyzes software written in the Function Block Diagram (FBD) language. The approach generates tests based on logical coverage criteria (e.g., MC/DC) by mapping FBD into UPPAAL.

Literature [33] introduces the AutoMOTGen toolset that allows for the mapping of Simulink/Stateflow models into the SAL framework and the model checking based test generation based on different logical coverage criteria.

AGEDIS [20] is an MBT toolset for component-based distributed systems. It integrates model and test suite editors, test simulation and debugging tools, test coverage and defect analysis tools, and report generators. Test models can be defined using UML class, state machine and object diagrams. Test generation uses the so-called TGV engine [26] and supports state and transition coverage.

Smartesting CertifyIt [28] is also an MBT toolset, which integrates editors for defining requirements and traceability, test adapters and test models. Test models are composed of UML class diagrams (for data description), state machines and object diagrams (initial states of executions), as well as Business Process Model and Notations (BPMN). Tests are generated with the CertifyIt Model Checker, which supports state, transition and transition-pair coverage criteria.

Our Gamma framework is unique from the aspect that it supports various composition modes [17] (including support for asynchronous communication), formally defined but configurable test coverage criteria with model reduction and slicing algorithms [16], and model checkers integrated via the XSTS formalism.

3 Extended Model-Based Testing Approach

This section first overviews the underlying *modeling languages* that support our extended model-based testing (MBT) approach for component-based reactive

systems. Next, the section details the *steps* (user activities and automated internal model transformations) that constitute the i) model design ii) formal verification and iii) test generation phases of our approach in Gamma.

- The **Gamma Statechart Language (GSL)** is a UML/SysML-inspired *configurable* formal statechart [19] language supporting different semantic variants of statecharts, e.g., different kinds of priorities for transitions [12].
- The **Gamma Composition Language (GCL)** is a composition language for the formal hierarchical composition of state-based (GSL) components according to multiple *execution* and *communication* (composition) modes [17]. It supports synchronous systems where components communicate with *sampled signals* and are executed *concurrently* (*synchronous-reactive*) or *sequentially* (*cascade*), as well as asynchronous systems where components communicate with *queued messages* and are executed *sequentially* (*scheduled asynchronous-reactive* [14]) or *in parallel* (*asynchronous-reactive*).
- The **Gamma Genmodel Language (GGL)** is a configuration language for configuring model transformations, e.g., to select the model checker for verification or the coverage criteria for test generation.
- The **Gamma Property Language (GPL)** is a property language supporting the definition of CTL* [7] properties and thus, the formal specification of requirements regarding (composite) component behavior.
- The **EXtended Symbolic Transition Systems (XSTS)** (Sect. 4) is a low-level formalism tailored to formal verification that supports the description of reactive behavior based on a set of *variables*, the values of which represent system states, and *transitions* that describe the possible state changes.
- The **Gamma Trace Language (GTL)** is a high-level trace language for reactive systems, supporting the description of execution traces, i.e., reached state configurations, variable values and output events in response to input events, time lapse and scheduling from the environment [17]. Such execution traces are also interpretable as *abstract tests* as the language also supports the specification of general assertions targeting the values of variables.

Figure 1 depicts the modeling languages, modeling artifacts and the model transformations of our MBT approach in Gamma. In the following, we overview the steps of the workflow, i.e., *model design*, *formal verification* and *test generation*; for more details regarding the formalization of the *test coverage criteria* and *model reduction* and *slicing* algorithms, we direct the reader to [16].

Model design The model design phase consists of three steps. As an optional step, *external component models*, i.e., statecharts created (Step 0) in integrated modeling tools (front-ends), are *imported* to Gamma by model transformations (Step 1) that map these models into GSL statecharts. Currently, the import of Yakindu [15], MagicDraw, SCXML [37], and XSML [14] models are supported. Next, the GSL models can be *hierarchically integrated* in GCL (Step 2) according to different (and potentially mixed) execution and communication modes [17].

Formal verification The integrated GCL model can be *formally verified* using a sequence of *automated model transformations* that map the model and

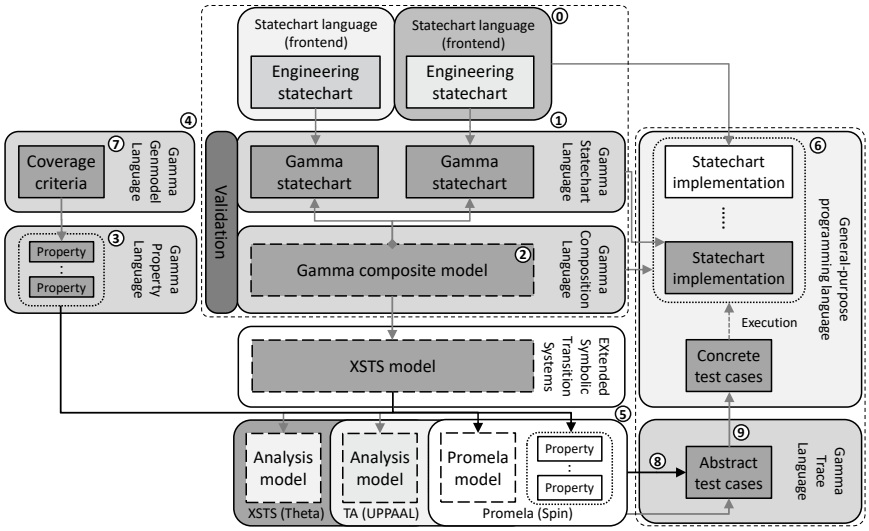

Fig. 1. Modeling languages and transformation chains of our approach in Gamma.

verifiable properties (specified in GPL – Step 3) into inputs of *model checker* back-ends via XSTS, i.e., UPPAAL, Theta or the newly integrated Spin. The mappings are configured using GGL (Step 4) to select back-ends, add optional constraints (e.g., scheduling) and set reduction and slicing algorithms [16]. The selected model checker *exhaustively explores* the model's state space with respect to the given property (Step 5) and potentially returns a diagnostic trace that is *back-annotated* to the GCL model, creating a representation in GTL.

Test generation If the verification results are satisfactory, *implementation* from the models can be derived manually or automatically using the code generators of Gamma or the integrated modeling front-ends (Step 6). *Integration tests* for the implementation can be *generated* based on the GCL model using the aforementioned formal verification facilities [10]. Test generation is driven by customizable dataflow-based, structural (model element-based) and behavior-based (interactional) *coverage criteria* (Step 7), which are *formalized* as reachability properties in GPL representing *test targets* (see [16] for their formalization), and control the model checkers to compute *test target (criterion) covering paths* during model traversal; model checking time can be limited by *timeouts* to discard uncoverable test targets. These paths, i.e., returned witnesses for satisfying the reachability properties, are represented as GTL execution traces (Step 8) and, in a testing context, are regarded as *abstract test cases* for the property based on which they are generated. *Optimization algorithms* are used to i) prevent starting model checking runs for already covered test targets and ii) remove unnecessary tests that do not contribute to the coverage of the specified criteria [16]. The abstract tests are concretized (Step 9) to execution environments (e.g., JUnit) by generating concrete calls to provide test inputs, time delay and schedule system execution, and then retrieve and evaluate outputs to check the *conformance* of the system model and implementation for these particular traces.

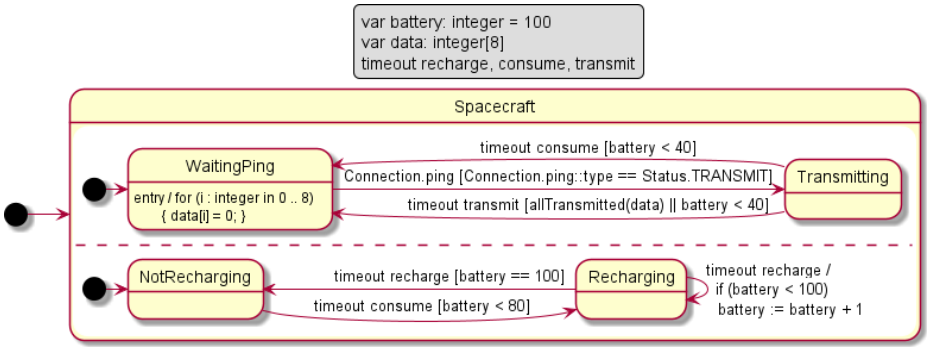

Fig. 2. Excerpt of the *spacecraft* component model.

For example, if we aim to cover with tests each *state node* in a GCL model, we can use the *state-coverage* keyword (with potential *include* and *exclude* constraints regarding the targeted state nodes) in a GGL *analysis model transformation* task besides selecting the model checker back-end for test generation. As a result, GPL reachability properties are generated for every targeted state node (in the form of `E F state` *`regionName.stateNodeName`*) besides the analysis model, which are, one by one, translated into the input property formalism of the selected model checker and checked on the generated analysis model. In case the targeted state node is reachable, the model checker returns a *diagnostic trace* consisting of a series of steps (alternating sequence of states and input events) leading to its coverage. This diagnostic trace is automatically back-annotated to GTL, creating an *abstract test case* that serves as a basis for test concretization.

Note that the architecture of the framework supports the *parallel running* of multiple model checkers (model checker portfolio) for test generation, this way, tackling the problem of finding the most suitable back-end for a given model.

4 EXtended Symbolic Transition Systems

The *symbolic transition systems* (STS) formalism [22] is a commonly used low-level representation, e.g., for hardware model checking. STS models consist of two SMT [2] formulas that describe the *set* of *initial states* and the *transition relation*. Our novel *EXtended Symbolic Transition Systems* (XSTS) formalism introduced here is built on top of STS, and serves as a formal representation for the behavior of component-based reactive systems.

The key contributions of our XSTS formalism compared to STS are as follows:

- XSTS introduces an *imperative layer* on top of SMT formulas, i.e., a set of *control structures* with operational semantics, having two advantages:
  1. It allows for the direct mapping of XSTS models into STS models, and thus, into SMT formulas (see the Appendix for details) for their verification by SMT-based model checkers. As a key feature, the mapping can exploit variable *substitution techniques* [35] to increase the efficiency of

verification by reducing the size and complexity of SMT formulas (e.g., by eliminating unreachable branches based on conditions) given to the back-end solvers. Control structures in the XSTS model facilitate the efficiency of model checking, as model mappings can utilize the capabilities of target model checkers that may have specific support for handling certain control structures (e.g., partial order reduction to handle parallel behavior), or peculiar abstraction for havoc structures (see below).
2. It supports the concrete execution of XSTS models (i.e., simulation functionalities) and code generation from XSTS models.

- XSTS introduces *annotations* to variables to support metadata attachments (e.g., to identify control variables). Annotations support *traceability* between source and target models (e.g., GCL and derived XSTS models) that can be leveraged during verification (e.g., special handling of the control variables).
- XSTS introduces *clock* variables to support time-dependent behavior.
- XSTS *partitions* the transition relation of STS to allow for distinguishing between the behavior of the environment and that of the modeled system. This feature can be leveraged during verification (back-annotation of diagnostic traces to higher-level models) and also during test generation by separating the steps of the environment and that of the modeled system in a generated execution trace, which is important when mapping to concrete test calls.

In the following, we introduce the elements of the XSTS language based on an example *spacecraft* statechart model (see Fig. 2) originally defined in [16] that transmits *data* to a *ground station* while managing its *battery*. Note that in general, the XSTS language has all the features necessary to capture in a semantic-preserving way the behavior of integrated GSL and GCL models.

Type declarations An XSTS model (see the left snippet of Fig. 3) begins with *custom type declarations* (**type** keyword) that contain *literals*, similarly to enum types in programming languages.

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

Transitions Model behavior is defined by three *transitions*, which are *atomic*, i.e., they are either *executed* in their *entirety* or *not at all*. The system's internal behavior is described by the **trans** transition (Lines 14-44), while the behavior of the system's environment is described by the **env** transition (Lines 48-58). The **init** transition (Lines 45-47) initializes the system. Regarding their execution order, the **init** transition is executed first, after which the **env** and **trans** transitions alternate. In our example, the **init** transition sets the statechart's initial state, while the **env** transition places a random message (more precisely, its identifier) in its queue. The **trans** transition pops a message from the queue, which is processed in both regions (see Sect. 6 for the details of queue handling).

Basic statements The detailed behavior of transitions is captured via *statements*. *Assign* statements (see Line 19) assign a value of its domain to a single variable. *Assume* statements (Line 34) act as guards; they can be executed only if their condition holds. *Havoc* statements assign a nondeterministically selected value of its domain to a variable (Line 55). *Local variable declarations* can be used to create *transient* variables that are only accessible in the scope they were created in and are not part of the system's state vector (Line 54).

Composite statements Composite statements contain other statements (operands), and can be used to describe complex control structures. *Sequences* are lists of statements that are executed sequentially; each statement operates on the result of the previous statement. *Choice* statements (see Line 33) model nondeterministic choices between multiple statements; only one branch is selected for execution, which cannot contain failing assumptions, i.e., if every branch contains failing assumptions, then the choice statement also fails. *Parallel* statements support the parallel execution of the operands (Lines 24-44). *If-else* statements are deterministic choices based on a condition (see Lines 25-31) with an optional *else* branch. XSTS also supports deterministic *for loops* over ranges (Line 30).

5 Transforming Asynchronous GCL Models into XSTS

The XSTS-Promela mapping (to be presented in Sect. 6) exploits the traits of the GCL-XSTS transformation (see the transformations related to Steps 4 and 5 in Sect. 3). Thus, we first overview the relevant parts of this automated transformation before moving onto the XSTS-Promela mapping in the next section.

Synchronous statechart components (defined in GSL) are the basic building blocks of GCL models (see Step 2 in Sect. 3). Asynchronous components are created from simple or composite synchronous ones (i.e., synchronous-reactive or cascade composite components) by wrapping them using a so-called *asynchronous adapter* [17]: an adapter maps *signals* used in the synchronous domain into *messages* and related *message queues* in the asynchronous domain and vice versa, i.e., the two representations of *events* in the two domains.

The transformation uses the same steps for every asynchronous adapter (detailed in [13]). Here, we overview the relevant parts of this procedure related to asynchronous communication based on the standalone *spacecraft* component and its (automatically generated) XSTS representation described in Fig. 3.

As for GSL statecharts, the XSTS elements corresponding to the *transitions* are created (Lines 24-43), e.g., *parallel* statements corresponding to orthogonal regions, containing *if-else* structures and *choice* statements corresponding to the transitions of state nodes (considering hierarchy) in the particular regions. Note that *literals* of a *custom type declaration* stored in a *control variable* correspond to the state nodes of a region.

As for composite (GCL) synchronous components, their execution and interactions are handled by (hierarchically) composing the created XSTS statements corresponding to the contained components, and defining shared variable based communication (corresponding to signal transmission) using *assign* statements.

```
type Status : { TRANSMIT, ... }
type Communication : { _Inact_,
     WaitingPing, Transmitting }
type Battery : { _Inact_, ... }
var ping : boolean = false
var ping_types : Status = TRANSMIT
ctrl var com : Communication = _Inact_
ctrl var bat : Battery = _Inact_
var batteryLevel : integer = 0
var recharge, transmit, consume : clock
// Message queue related variables
@QueueSize var len : integer
@Queue var msgQueue : [integer] -> integer
@Queue var argQueue : [integer] -> Status
trans {
 if (len > 0) {
  local var id: integer := msgQueue[0];
  msgQueue[0] := 0; // Message pop
  if (id == 1) {
   ping := true;
   ping_types := argQueue[0];
   argQueue[0] := 0; // Arg. pop
  } ... // Potentially other msgIds
  len := len - 1;
  par {
   if (com == WaitingPing && ping &&
     ping_types == TRANSMIT) {
    com := Transmitting;
   } else if (... // Second transition
   else if (com == Transmitting && ... <=
     transmit) {
    com := WaitingPing;
    for i from 0 to 7 do { data[i] := 0; }
   }
  } and {
   choice {
    assume (bat == NotRecharging && ... <=
      consume &&  batteryLevel < 80);
    bat := Recharging;
    recharge := 0;
   } or {... // Second transition
   } or {
    assume (bat == Recharging && ... <=
      recharge && batteryLevel == 100);
    bat := NotRecharging;
   } or {
    assume !(...); // Else branch
   } } ... // Input event clearing
 } }
init { ... // Variable initializations
 com := WaitingPing; // Initial sates
 bat := NotRecharging; ... } // Entries
env {
 if (len <= 0) {
  local var msgId : integer;
  havoc msgId;
  if (0 < msgId && msgId <= 1) {
   msgQueue[len] := msgId;
   local var argument : Status;
   havoc argument;
   argQueue[len] := argument;
   len := len + 1; }
 } }
```

```
mtype:Status = { Status_TRANSMIT, ... }
mtype:Communication = { C_Inact_,
     WaitingPing, Transmitting }
... // Proc-sync related variables
chan chan_parallel_0 = [0] of { bit };
chan chan_parallel_1 = [0] of { bit };
// Metadata variables
byte flag = 0; bit isStable = 0; //
proctype Parallel_0() {
 if
  :: (com == WaitingPing && ping &&
     ping_types == Status_TRANSMIT) -> com
     = Transmitting;
  :: else -> if
   ... // Second transition
    :: else -> if
      :: (com == Transmitting && ... <=
     transmit) -> com = WaitingPing;
       for (i : 0 .. 7) { data[i] = 0; }
      :: else
     fi;
   fi;
 fi;
 chan_parallel_0 ! 1; // Finished execution
}
proctype Parallel_1() {
 if
  :: if :: (bat == NotRecharging &&
     batteryLevel < 80); fi;
  bat = Recharging;
  recharge = 0;
  :: if :: ... // Second transition
  :: if :: (bat == Recharging && ... <=
     recharge && batteryLevel == 100); fi;
  bat = NotRecharging;
  :: else;
 fi;
 chan_parallel_1 ! 1; // Finished execution
}
proctype EnvTrans() { (flag > 0);
 ENV: isStable = 1; // Verification may end
 atomic {
  isStable = 0;
  if // Mapping a havoc statement
   :: msgId = 0;
   :: msgId = 1;
  fi;
  if :: (0 < msgId && msgId <= 1) ->
   ... // Mapping-specific array handling
  fi;
 }
 TRANS: atomic { flag = 2; // TRANS is next
  run Parallel_0(); run Parallel_1();
  bit msg_parallel_0 = 0; // Process sync
  chan_parallel_0 ? msg_parallel_0;
  chan_parallel_1 ? msg_parallel_0;
  ... // Input event clearing
  flag = 1; }; // ENV is next
 goto ENV;
}
init { ... // Variable initializations
 { run EnvTrans(); flag = 1; } // Run proc.
}
```

Fig. 3. XSTS and Promela representations of the *spacecraft* component of Fig. 2.

```
1  int queue[Capac]; // Queue array
2  int len = 0; // Size variable
3  // Nonempty
4  len > 0
5  // Append
6  queue[len] = value; len = len + 1;
7  // Peek
8  int peekVariable = queue[0];
9  ... // Referencing peekVariable
10 peekVariable = 0; // Resetting "local" var
11 // Pop
12 queue[0] = queue[1]; queue[1] = queue[2];..
13 queue[len - 1] = 0;
14 len = len - 1;
```

```
1  chan queue = [Capac] of { int }; // Queue
2  // Nonempty
3  len(queue) > 0
4  // Append
5  queue ! value;
6  // Peek
7  int peekVariable;
8  queue ? <peekVariable>;
9  ... // Referencing peekVariable
10 peekVariable = 0; // Resetting "local" var
11 // Pop
12 int popVariable;
13 queue ? popVariable;
14 popVariable = 0; // Resetting "local" var
```

Fig. 4. XSTS-Promela mapping's two supported message queue representation modes.

For the asynchronous adapters defined in GCL, the related XSTS constructs "wrap" the statements representing synchronous behavior by introducing and handling additional variables related to the representation of message queues (see Lines 15-23). For a single message queue in an asynchronous adapter, the transformation introduces one or more annotated *array* variables (denoting that the variable represents a queue) depending on whether the queue stores only non-parameterized messages (*msgQueue* array, Line 12) or also message parameter values, i.e., payload (one or more *argQueue* arrays for every *parameter type* to enable storing each parameter value of each message; Line 13). In addition, an integer variable with an annotation is introduced (*len*, see Line 11) storing the number of messages present in the queue (the annotation denotes that the variable stores the size of a queue). In general, every message type stored in the queue is assigned an integer identifier that is appended to the *msgQueue* array, modeling the append of the message instance to the corresponding message queue (see Line 53). In case the message is parameterized, the parameter value is appended to the corresponding *argQueue* array (see Line 56).

Asynchronous message handling is defined using the *nonempty*, *peek* and *pop* message queue operations (the Promela representation of these operations is summarized in the left snippet of Fig. 4). An *if* statement is created that checks whether the *msgQueue* variable corresponding to the message queue is *nonempty* ($len > 0$ condition, see Line 15), and if so, the stored message identifier is retrieved (*peek* and *pop*, see Lines 16-17) and based on it, the corresponding input event (signal) variable is set to true (Line 19). Potentially, the parameter values are also loaded from the *argQueues* to the corresponding input parameter variables (*peek* and *pop*, see Lines 20-21). Finally, the constructs representing the synchronous behavior are wrapped into the created *if* statement.

Regarding the verification of the emergent XSTS models, the different model checkers (i.e., corresponding mappings) integrated into Gamma support slightly different subsets of the XSTS language, as well as different property languages, i.e., supported subsets of the GPL language (see Table 1): UPPAAL fully supports the verification of time-dependent behavior with a restricted CTL property language, whereas Spin fully supports the verification of parallel behavior (see

Table 1. Overview of features of the mappings and underlying model checkers supported by our MBT approach in Gamma. ✓= full support; ✓̸= experimental

Model checker (back-end)	Model representation	Parallel behavior	Timed behavior	Asynchronous communication	Property language
Theta	XSTS	✓̸	✓̸	✓	Reachability
UPPAAL	Timed automata		✓	✓	Restricted CTL
Spin	Promela process models	✓		✓	LTL

parallel statements in Sect. 6) and LTL [7]; in turn, Theta has experimental support for these features and supports only reachability properties. Nonetheless, every model checker supports the XSTS constructs used for asynchronous communication as presented in this section, as well as the specification of reachability properties to capture test targets in our MBT approach.

6 Mapping XSTS Models into Promela

The integration of Spin into the Gamma framework relies on a semantic-preserving mapping between XSTS and Promela. The mapping distinguishes XSTS elements (see Sect. 4) that i) *do have* or ii) *do not* have a *direct semantic equivalent* in Promela, as well as iii) constructs related to *asynchronous communication*. The only XSTS element whose semantic-preserving mapping is not supported is *clock variable*, as Promela does not support time-dependent behavior: for such variables, the mapping introduces integer variables in the Promela model and "discretizes" component execution according to user-defined constraints.

In the case of i), the mapping is simple and creates a single Promela element with the same semantics, as presented in Table 2. Note that several constructs are optimized, e.g., the mapping resets local variable declarations at the end of their declaring scope and uses *d_step* for the sequences of assignments. Contrarily, in the case of ii) and iii), multiple Promela elements are created to preserve the semantics of the original XSTS elements and asynchronous communication.

In the following, we consider XSTS constructs related to ii) and iii). First, we present the mapping of XSTS *transitions*, *havoc* and *parallel* statements based on the *spacecraft* model represented in Fig. 3. Then, we describe asynchronous communication supported by two message queue representation modes.

Mapping elements without a direct semantic equivalent The *init* transition is mapped into Promela's *init* process (see Lines 55-57 in the Promela code), which,

Table 2. XSTS elements and their *direct semantic equivalent* element(s) in Promela.

XSTS	Promela
boolean, **integer**, **custom**, **array** type	**bit**, **int**, **mtype**, **array** type
(*local*) **variable declaration**	(*local*) **variable declaration** (+ *resets*)
assume statement	**boolean** expression (in **if** construct)
assignment statement	**assignment** statement
sequence (of statements)	**sequence** (of statements) (+ *d_step*)
choice statement	*nondeterministic multiary* **if** construct
if-else statement	*binary* **if** construct with **else**
loop statement	**for** deterministic iteration statement

after executing the mapped initialization statements, runs the *EnvTrans* process (Line 34). The *EnvTrans* process comprises (in a *cycle*) the statements of the *env* and *trans* transitions; the corresponding statements are wrapped in *atomic* blocks and labeled *ENV* (Line 35) and *TRANS* (Line 46). The states of execution are indicated by the *isStable* and *flag* (meta)variables (Line 7): the former is true iff *trans* has finished, but the execution of *env* has not started yet (needed to specify valid *end states* in property specifications – Line 35), while the latter encodes which transition is under execution: 0 – *init*, 1 – *env* and 2 – *trans* (needed for back-annotation – Lines 46 and 52).

Havoc statements describe nondeterministic assignments from the targeted variable's domain and thus, they are mapped into nondeterministic multiary *if* selection constructs whose *options* describe the possible values based on the variable's type (see Line 38). For *boolean* variables, *true* and *false* (0 and 1) values, for *custom* types, the declared *literals* are included. *Integer* variables are handled only if their domain is *restricted*, e.g., in the case of stored message identifiers; otherwise this construct is not supported in the Promela mapping.

Parallel statements (i.e., their operands; see Lines 8 and 22) are mapped into Promela processes and synchronization constructs. First, the *local variables* (if any) referenced from the operands are identified, which are mapped into *parameter declarations* in the corresponding processes (*assignments* to local variables are not supported). Next, *synchronization channels* are created (Lines 4-5) to allow for synchronization between the caller and the parallelly running processes: the caller waits for the started processes to finish execution (Lines 47-50).

Mapping elements of asynchronous communication As for asynchronous communication, the XSTS-Promela mapping builds on the traits of the GCL-XSTS transformation (see Sect. 5), in particular, the annotated array variables (*msgQueue* and *argQueue*) and *len* variables, jointly representing the message queues of asynchronous adapters. As a configuration option, the mapping features two (fixed capacity) message queue representation modes based on i) *arrays* and ii) native *asynchronous channels* in Promela. Note that the former can be considered as a straightforward mapping (resembling the one to UPPAAL), whereas the latter serves as an experiment in the context of Promela, potentially being more efficient (see Sect. 7). Figure 4 summarizes how the *nonempty*, *append*, *peek*, *pop* and *size* queue handling operations are represented in the two message queue representation modes. Note that Fig. 3 uses the array-based mode, but the other version could be easily reproduced based on Fig. 4.

In the former case, the mapping is straightforward: the XSTS array variables and *len* variables are mapped into Promela array variables and integer variables. The statements that refer to these variables and correspond to message queue operations are also mapped according to the rules presented in Table 2.

The latter mode uses the *asynchronous channel* construct of Promela that natively supports the *size* queue operation using the built-in *len* function, disregarding the XSTS integer variable *len*. To support the *nonempty*, *append*, *peek* and *pop* message queue operations, the mapping traverses the XSTS model and identifies the corresponding statements (i.e., the constructs of the left snippet of

Fig. 4) based on variable annotations and pattern matching, which are mapped to native message queue related constructs in Promela (right snippet of Fig. 4).

7 Practical Evaluation

This section evaluates our extended MBT approach, focusing on the features of the model mappings via the XSTS formalism and their performance in the context of the integrated model checker back-ends, Theta, UPPAAL and Spin.

We conduct this evaluation in alignment with the needs of our industrial partner that develops railway control systems in the context of customized MBSE and CBSE approaches. During development, our partner must conform to safety standards (cf. EN 50128 [5]), which require (among others) integration test generation based on these models to check the system implementation. Thus, our partner has interest in conducting this task in an automated and efficient way.

In regard to test generation, we already showed the feasibility of our approach for synchronous models in [16] using the UPPAAL and Theta model checkers. Nevertheless, our partner uses different composition semantics at different hierarchy levels of a system model (considering different deployment modes), and is interested in generating tests for components with asynchronous communication, too. Accordingly, our partner needs information regarding the characteristics of the supported model checkers and message queue representations.

Thus, we formulated the following research questions (RQ) for our evaluation: how efficient are the integrated model checkers of Gamma for *test generation*, in terms of *generation time* and *generated test set size*, on

RQ-1. *synchronous* models (*shared variable* based communication), and
RQ-2. *asynchronous* models (*message queue* based communication) using a i) native *channel* based (relevant only in the case of Spin), and ii) *array*-based mapping of GCL/XSTS message queues?

Models We evaluated the RQ on two system models received from our industrial partner, namely *railway signaller subsystem* (RSS) and *railway interlocking system* (RIS), each corresponding to the characteristics of a specific RQ.[2]

RSS comprises the model of a subsystem used in railway traffic control systems [11]. It builds on statechart components (two *antivalence checkers* connected to a *signaller*) communicating in a synchronous (signal-based) way with many interaction points (dispatch/reception of signals in different states). Thus, it is a relevant model for generating tests that check interactions. RSS consists of 26 state nodes, 97 transitions and 5 variables.

The RIS model [11] defines an industrial communication protocol used in RIS, and comprises three components defined in the proprietary XSML language (integrated to Gamma in [14]), namely *control center*, *dispatcher* and *object handler*. The components are executed sequentially, and communicate with their

[2] Model descriptions and measurement results can be found at `https://github.com/ftsrg/gamma/tree/v2.9.0/examples/`, as well as [16] (RSS) and [14] (RIS).

messages stored in local message queues; the *control center* and *object handler* can communicate only via the *dispatcher*. Thus, testing interactions based on messages besides covering state nodes and transitions is relevant in this model's context. The model contains 38 state nodes, 118 transitions and 23 variables.

Measurement settings We used different *composition modes* in the different models to capture the components' expected execution and communication modes. For the RSS model, we used the *cascade* and *synchronous-reactive* composition modes [17]. For RIS, we used the *scheduled asynchronous-reactive* mode [14] and the capacity of 4 for internal message queues between components. The message queues storing external messages had a capacity of 1 as only the head of the queue was of interest during execution (one message can be processed in an execution cycle). Regarding verification and test generation settings, we used all *model reduction* and *slicing* techniques, and *test optimization* algorithms of Gamma [16,14] to achieve the best results our framework can provide.

Each measurement was run *five* times (we calculated their *median*) on the following configuration: *Intel Core (TM) i5-1135G7 @ 2.40GHz, DDR4 16GB @ 3.2GHz, SSD 500GB*. The model checkers were run with the following arguments:

- **Theta** (**DX**): *java -jar theta –domain PRED_CART –refinement SEQ_ITP* – predicate abstraction based algorithms (most suitable for XSTS);
- **UPPAAL** (**XU**): *verifyta -t0* – set to generate "some trace" (with default BFS traversal) as predictably, this is the "fastest" option (the "shortest trace" option, according to our preliminary measurements, results in "slightly" shorter generated traces but "significantly" longer generation time);
- **Spin**: we used three different sets of arguments (settings):
  1. **XP**: *spin -search -I -m250000 -w32* – DFS *approximate* iterative shortening, increased bound (larger than the state space's estimated maximum "diameter") and increased hash size for better performance: compared to the next setting, this one provides longer (not optimal, but still *sound*) traces but shorter generation time (see exact results later);
  2. **XP-i**: *spin -search -i -m250000 -w32* – *non-approximate* iterative shortening algorithm variant of the previous option;
  3. **XP-B**: *spin -search -bfs -w32* – BFS traversal of the state space.

Regarding the Spin-related arguments, the XP setting *worked* for *both models*; the other two worked only for the RSS model. For RIS, the XP-B option *ran out of memory* within 10 seconds, whereas XP-i *ran out* of the *300-second time limit* (timeout) for certain model-property pairs.

Addressing RQ-1 Table 3 shows test generation results for the RSS model aiming at covering each *interaction* [14] between the *antivalence checkers* (event raises) and *signaller* components (execution of transitions triggered by the corresponding event) in two model variants using the *cascade* and *synchronous-reactive* composition modes. As illustrated, there were 49 test targets (captured using injected boolean variables and reachability properties for each interaction [16] in the form of `E F var eventRaised and var correspondingTransitionFired`)

Table 3. Number of *test targets*, *generated tests* and *steps*; median end-to-end *test generation time* and *average test generation time* for a *single test target* for full *interaction* coverage in the *cascade* and *synchronous-reactive* RSS model.

	Cascade	**Synchronous**
#Test targets	49	49
#Generated tests (DX/XU/XP/XP-B/XP-i) *#Steps in tests* (DX/XU/XP/XP-B/XP-i)	9/7/6/6/6 41/30/172/27/27	-/7/5/6/6 -/37/600/33/33
Σ**T** (DX/XU/XP/XP-B/XP-i) (s)	467/54/179/188/340	-/102/174/220/1102
$\overline{\text{T}}$ (DX/XU/XP/XP-B/XP-i) (s)	9.5/1.1/3.7/3.8/6.9	-/2.1/3.6/4.5/22.5

that could potentially be covered by the Theta (DX), UPPAAL (XU) and Spin (XP, XP-B and XP-i) back-ends.

Regarding *test generation time*, Theta was the slowest in the cascade model variant and could not handle the more complex synchronous-reactive variant. As for UPPAAL and Spin, the former was significantly faster (70% and 42% for XP) considering end-to-end generation time. Nonetheless, Spin seemed to scale better as the synchronous-reactive model variant did not pose a greater challenge for the XP setting compared to the cascade variant – in contrast to UPPAAL.

Regarding the *generated test size*, the XP-B and XP-i Spin settings generated the smallest test sets (least number of summed steps in tests), even though the former setting was much more efficient in terms of *generation time*. UPPAAL generated 10% longer traces on average, as expectable, considering that it uses a BFS-based model traversal mode aimed at "some trace." In addition, Theta (in the cascade variant) returned a 37% larger test set compared to UPPAAL. The XP setting returned significantly larger test sets compared to the other model checkers, even though it used the *approximate DFS iterative shortening* algorithm. In turn, these longer generated tests were utilized by the test optimization algorithms to cover the same interactions with *fewer test cases*.

To answer RQ-1, the experiment showed that even though Spin is slightly slower in terms of test generation (verification) time than its fastest counterpart (UPPAAL), it seems to be able to handle more complex interactions more efficiently, and thus, it has advantages for such models in the framework. Regarding the generated test set size, the length of the traces returned by Spin differ largely depending on the used settings. The results also show that there is a trade-off between the length of generated traces and generation time.

Addressing RQ-2 Table 4 shows test generation results aiming at covering each *state node*, *transition* and *interaction* in the asynchronous RIS model using the XU (with *array-based* mapping of message queues into UPPAAL), and the *native channel-based* (XP-N) and *array-based* (XP-A) mappings of GCL/XSTS message queues into Promela. The table does not include results for the DX mapping as Theta was unable to handle this model.

The data show that *test generation time* increased for each verification backend as the test coverage criteria got finer. This phenomenon can be explained by the complexity of annotated model elements capturing the criteria, and the increasing number of test targets [16]. Surprisingly, Spin was on average 43%

Table 4. Number of *test targets*, *generated tests* and *steps*; median end-to-end *test generation time* and *average test generation time* for a *single test target* for full *state node*, *transition* and *interaction* coverage in the integrated RIS model.

	State	**Transition**	**Interaction**
#Test targets	38	118	387
#Generated tests (XU/XP-N/XP-A)	4/5/5	26/34/34	22/24/24
#Steps in tests (XU/XP-N/XP-A)	30/119/119	230/870/870	240/808/808
Σ**T** (XU/XP-N/XP-A) (s)	243/140/135	950/1777/1653	5377/8315/7840
$\overline{\mathrm{T}}$ (XU/XP-N/XP-A) (s)	6.4/3.7/3.6	8.1/15.1/14.0	13.9/21.4/20.3

faster than UPPAAL in the case of *state node* coverage even though the result was reversed for *transition* and *interaction* coverage (45% and 34% slower on average). Regarding the two message queue representation modes, the *array-based* one was slightly faster in each case, having a 5% advantage on average.

As for the *size* of the *generated test sets*, the results were similar to that of the RSS model: Spin returned significantly longer traces compared to UPPAAL due to the underlying algorithms; however, in this case the number of generated tests was also larger. Also, there was no difference in the two message queue representation modes in this regard; Spin returned the same traces in each case.

To answer RQ-2, the experiment showed that Spin is an efficient and useful element of the framework and can provide additional flexibility in the case of different models and test coverage criteria. Regarding our message queue representation modes, our native *asynchronous channel* based solution does not bring benefits compared to our *array-based* solution. Regarding generated test set size, the results are very similar to that of RQ-1: Spin with the XP setting returns significantly longer diagnostic traces, resulting in larger test sets.

8 Conclusion and Future Work

In this paper, we presented an MBT approach for distributed (control-oriented) reactive systems with asynchronous communication. Our approach builds on the Gamma framework and features precise statechart and composition languages for component design and their integration, and hidden formal methods (model checkers) for model-based test generation. As a novelty, we integrated the open source Spin model checker to the Gamma framework via the *EXtended Symbolic Transition Systems* formalism, a verification-oriented low-level representation of reactive behavior. Our evaluation showed that Spin is efficient at generating tests and can complement the other integrated model checkers for different models.

For future work, we plan to examine the capabilities of Spin for models with parallel behavior, and how we can exploit its property language supporting linear temporal logic (LTL) [7] to capture more sophisticated test targets – two features not supported by the other integrated model checkers.

Acknowledgements. We would like to thank the anonymous reviewers for their thorough and constructive feedback. This work was partially supported by New National Excellence Program of the Ministry for Innovation and Technology,

ÚNKP-23-4-I. Project no. 2019-1.3.1-KK-2019-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.3.1-KK funding scheme.

Appendix: XSTS-STS Mapping

Symbolic transition system [18] models consist of two SMT [2] formulas. The *set* of *initial states* are described by a formula over the model variables, e.g., $x = 0$. The *transition relation* is described by a formula over the primed and unprimed versions of the model variables; the primed versions refer to the newly assigned values of the variables, e.g., $x' = x + 1$, meaning that x variable's value gets incremented by 1. To allow multiple assignments to the same variable in the transition, we allow variables to appear with more than one prime sign and annotate the transition relation with a function (*nextIndex*), describing which primed version corresponds to the variable's value in the next state after the transition; e.g., if the transition formula is $x' = 1 \wedge x'' = 2$ and the *nextIndex* function is $x \to 2$, then the x variable's value will be 2 after the transition.

Table 5 shows the mapping between the statements of the XSTS language and their equivalent STS transition formulas through examples. An *assignment* simply asserts that the next value of the left-hand-side is equal to the value of the right-hand-side. *Assumptions* translate directly into SMT formulas, while a *havoc* statement introduces the next value of the variable, but does not constrain it (therefore it can have any value from its domain).

Composite statements compose the mapping of their constituent statements. A *sequence* maps to a conjunction of its statements, but each statement will use the current primed versions of the variables. A *choice* statement requires an unconstrained temporary variable – the assignment of this variable by the solver will also determine which branch may be true (at most exactly one). An *if-else*, on the other hand, uses implication to select the branch that is asserted to be true, the else statement meaning the negation of the conjunction of all other branch conditions. A *for loop* is unfolded into a sequence, but this is done by the model checking algorithm, so the number of iterations may depend on the current state (but not the execution of the statement itself).

Table 5. XSTS elements and their equivalent STS formulas.

Statement	**XSTS example**	**STS equivalent**	**nextIndex**
Assignment	$x := y$	x' = y	$x \to 1, y \to 0$
Assumption	**assume** $x > 5$	$x > 5$	$x \to 0$
Havoc	**havoc** x	*true* (x' is unconstrained)	$x \to 1$
Sequence	$x := y$ **assume** $x > y$ $y := 3$	$x' = y \wedge$ $x' > y \wedge$ $y' = 3$	$x \to 1, y \to 1$
Choice	**choice** { $x := y$ } **or** { $x := y + 1$ }	$(temp = 0 \ \wedge \ x' = y) \vee$ $(temp = 1 \ \wedge \ x' = y + 1)$	$x \to 1, y \to 0$
If-else	**if** $(x > y)$ { $x := 0$ } **else** { $x := 1$ }	$(x > y \Rightarrow x' = 0) \wedge$ $(x \leq y \Rightarrow x' = 1)$	$x \to 1, y \to 0$
Loop	**for** i **from** 0 **to** 1 **do** { $x := i$ }	$(i' = 0 \ \wedge \ x' = i') \wedge$ $(i'' = 1 \ \wedge \ x'' = i'')$	$x \to 2, i \to 2$

References

1. Ambrosi, G., Bartocci, S., et al., L.B.: The electronics of the high-energy particle detector on board the CSES-01 satellite. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **1013**, 165639 (2021). https://doi.org/10.1016/j.nima.2021.165639
2. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of model checking, pp. 305–343. Springer (2018)
3. Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software **28**(3), 41–48 (2011). https://doi.org/10.1109/MS.2011.27
4. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems. p. 125–126. QEST '06, IEEE Computer Society, USA (2006). https://doi.org/10.1109/QEST.2006.59
5. Boulanger, J.L.: CENELEC 50128 and IEC 62279 standards. John Wiley & Sons (2015)
6. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)
7. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. J. ACM **33**(1), 151–178 (1986). https://doi.org/10.1145/4904.4999
8. Enoiu, E.P., Čaušević, A., Ostrand, T.J., Weyuker, E.J., Sundmark, D., Pettersson, P.: Automated test generation using model checking: an industrial evaluation. International Journal on Software Tools for Technology Transfer **18**(3), 335–353 (2016). https://doi.org/10.1007/s10009-014-0355-9
9. Ferrari, A., Mazzanti, F., Basile, D., ter Beek, M.H.: Systematic evaluation and usability analysis of formal methods tools for railway signaling system design. IEEE Transactions on Software Engineering **48**(11), 4675–4691 (2022). https://doi.org/10.1109/TSE.2021.3124677
10. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Software Testing, Verification and Reliability **19**(3), 215–261 (2009). https://doi.org/10.1002/stvr.402
11. Golarits, Z., Sinka, D., Jávor, A.: Proris — a new interlocking system for regional and moderate-traffic lines. SIGNAL+DRAHT - SIGNALLING & DATACOMMUNICATION (114), 28–36 (2022)
12. Graics, B.: Documentation of the Gamma Statechart Composition Framework v0.9. Tech. rep., Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems (2016), https://tinyurl.com/yeywrkd6
13. Graics, B.: Mixed-semantic composition and verification of reactive components. Tech. rep., Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems (2023), `https://tinyurl.com/2p9dae58`
14. Graics, B., Majzik, I.: Integration test generation and formal verification for distributed controllers. In: Renczes, B. (ed.) Proceedings of the 30th PhD Minisymposium. Budapest Univ. of Technology and Economics, Department of Measurement and Information Systems (2023). https://doi.org/10.3311/minisy2023-001
15. Graics, B., Molnár, V.: Formal compositional semantics for Yakindu statecharts. In: Pataki, B. (ed.) Proceedings of the 24th PhD Mini-Symposium. p. 22–25. Budapest Univ. of Technology and Economics, Department of Measurement and Information Systems, Budapest, Hungary (2017)

16. Graics, B., Molnár, V., Majzik, I.: Integration test generation for state-based components in the Gamma framework. Preprint (2022), https://tinyurl.com/4dhubca4
17. Graics, B., Molnár, V., Vörös, A., Majzik, I., Varró, D.: Mixed-semantics composition of statecharts for the component-based design of reactive systems. Software and Systems Modeling **19**(6), 1483–1517 (2020). https://doi.org/10.1007/s10270-020-00806-5
18. Hajdu, A., Tóth, T., Vörös, A., Majzik, I.: A configurable CEGAR framework with interpolation-based refinements. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components and Systems, Lecture Notes in Computer Science, vol. 9688, pp. 158–174. Springer (2016). https://doi.org/10.1007/978-3-319-39570-8_11
19. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
20. Hartman, A., Nagin, K.: The AGEDIS tools for model based testing. ACM Sigsoft Software Engineering Notes **29** (2004). https://doi.org/10.1145/1007512.1007529
21. Heineman, G.T., Councill, W.T.: Component-based software engineering. Putting the Pieces Together, Addison Wesley (2001). https://doi.org/10.5555/379381
22. Henzinger, T.A., Majumdar, R.: A classification of symbolic transition systems. In: Reichel, H., Tison, S. (eds.) STACS 2000. pp. 13–34. Springer Berlin Heidelberg (2000)
23. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 1st edn. (2011)
24. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
25. Huang, L.: The past, present and future of railway interlocking system. In: 2020 IEEE 5th International Conference on Intelligent Transportation Engineering (ICITE). pp. 170–174 (2020). https://doi.org/10.1109/ICITE50838.2020.9231438
26. Jéron, T., Morel, P.: Test generation derived from model-checking. In: International Conference on Computer Aided Verification. pp. 108–122. Springer (1999)
27. Ke, X., Sierszecki, K., Angelov, C.: COMDES-II: A component-based framework for generative development of distributed real-time control systems. In: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 199–208 (2007). https://doi.org/10.1109/RTCSA.2007.29
28. Legeard, B., Bouzy, A.: Smartesting CertifyIt: Model-based testing for enterprise IT. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. pp. 391–397 (2013). https://doi.org/10.1109/ICST.2013.55
29. Li, J., Post, M., Wright, T., Lee, R.: Design of attitude control systems for cubesat-class nanosatellite. J. Control Sci. Eng. **2013** (jan 2013). https://doi.org/10.1155/2013/657182
30. Li, W., Le Gall, F., Spaseski, N.: A survey on model-based testing tools for test case generation. In: Itsykson, V., Scedrov, A., Zakharov, V. (eds.) Tools and Methods of Program Analysis. pp. 77–89. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-71734-0_7
31. Lukács, G., Bartha, T.: Formal modeling and verification of the functionality of electronic urban railway control systems through a case study. Urban Rail Transit **8** (2022). https://doi.org/10.1007/s40864-022-00177-8
32. Martinez, S., Pereira, D.I.D.A., Bon, P., Collart-Dutilleul, S., Perin, M.: Towards safe and secure computer based railway interlocking systems. International Journal of Transport Development and Integration **4**(3), 218–229 (2020)

33. Mohalik, S., Gadkari, A.A., Yeolekar, A., Shashidhar, K., Ramesh, S.: Automatic test case generation from Simulink/Stateflow models using model checking. Softw. Test. Verif. Reliab. **24**, 155–180 (2014). https://doi.org/10.1002/stvr.1489
34. Molnár, V., Graics, B., Vörös, A., Majzik, I., Varró, D.: The Gamma Statechart Composition Framework. In: 40th International Conference on Software Engineering (ICSE). pp. 113–116. ACM, Gothenburg, Sweden (2018). https://doi.org/10.1145/3183440.3183489
35. Mondok, M.: Efficient abstraction-based model checking using domain-specific information. Tech. rep., Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems (2021), `https://tinyurl.com/yh4b8w98`
36. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer (1992)
37. Radnai, B.: Integration of SCXML state machines to the Gamma framework. Tech. rep., Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems (2022), `https://tinyurl.com/4mmtsw7v`
38. Shankar, N.: Symbolic analysis of transition systems? In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) Abstract State Machines - Theory and Applications. pp. 287–302. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
39. Sztipanovits, J., Bapty, T., Neema, S., Howard, L., Jackson, E.: OpenMETA: A model- and component-based design tool chain for cyber-physical systems. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) From Programs to Systems. The Systems perspective in Computing. pp. 235–248. Springer (2014). https://doi.org/10.1007/978-3-642-54848-2_16
40. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: a framework for abstraction refinement-based model checking. In: Stewart, D., Weissenbacher, G. (eds.) Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257
41. Zhou, J., Hu, Q., Friswell, M.I.: Decentralized finite time attitude synchronization control of satellite formation flying. Journal of Guidance, Control, and Dynamics **36**(1), 185–195 (2013). https://doi.org/10.2514/1.56740