Integration Test Generation and Formal Verification for Distributed Controllers

Bence Graics, István Majzik
Budapest University of Technology and Economics,
Department of Measurement and Information Systems
Budapest, Hungary
Email: {graics, majzik}@mit.bme.hu

***Abstract*—Software-intensive distributed controllers are becoming increasingly prevalent, among others, also in railway interlocking systems (RIS). As such systems carry out critical tasks, their systematic verification and testing are a must, which can be supported by formal methods. This paper presents a verification and testing approach for a distributed RIS subsystem using hidden formal methods. The subsystem's functional behavior is modeled using statechart components defined in a high-level UML-based modeling language, which are integrated according to sound execution and interaction semantics defined by the RIS protocol. The emergent model is automatically mapped into input formalisms of model checker back-ends. Integration tests for the system implementation are derived according to various model-based coverage criteria using the model checker back-ends and generated properties. The approach is implemented in our open source Gamma Statechart Composition Framework.**

***Index Terms*—MBSE, collaborating statecharts, hidden formal methods, model checking, test generation, integrated tool suite**

I. Introduction

Software-intensive programmable controllers are getting more and more prevalent in critical infrastructure, e.g., in railway interlocking systems (RIS). Such systems are generally embedded into their dynamic environment and must coordinate multiple subsystems/components to carry out complex tasks in response to external commands or environmental changes.

The distributed nature of these systems encumbers their design, necessitating precise means to describe the integration of system components, including their execution and communication, in addition to their standalone behavior. Moreover, as these systems carry out critical tasks, the automated formal verification of their design and the testing of their implementation are a must in the development process.

Model-based and component-based systems engineering (MBSE and CBSE) [1], [2] approaches promote the use of reusable models and components based on high-level modeling languages. However, they usually do not provide sophisticated tool-centric means for the automated verification of the design artifacts or the testing of the system implementation, either due to informal model descriptions or the lack of sound and efficient verification methods.

These issues are also prevalent in the railway domain: [3] and [4] argue that the main barriers that hinder the wide adoption of verification-oriented MBSE/CBSE approaches stem from the lack of traceability and process integration between the high-level models and verification back-ends.

This work was supported by the ÚNKP-22-3 New National Excellence Program of the Ministry for Innovation and Technology.

This paper offers solutions to these shortcomings and presents a verification and test generation approach (based on hidden formal methods) for distributed controllers, adapted to the design language and integration semantics of a specific RIS design. The approach relies on the proprietary EXtended State Machine Language (XSML) used in a railway project to model the functional behavior of system components. The models are transformed into the statechart language (GSL) [5] of our Gamma Statechart Composition Framework [6], a framework for the component-based design and verification of reactive systems. The models are integrated in Gamma's composition language (GCL) [7] based on sound execution and communication semantics in accordance with the system specification. The composite model is then mapped into input formalisms of model checker back-ends to support the exhaustive verification of its behavior and generate integration tests for the implementation based on various coverage criteria. The mappings feature model reduction and slicing algorithms to support industrial-scale systems.

Similar frameworks have been introduced in [8], [9] and [10] for the verification of and test generation for component-based reactive systems. However, these approaches rely on commercial tools, hindering their extensibility, and do not provide flexibility in terms of integration semantics, contrary to our approach building on the open source Gamma framework.

Our novel contributions are *(1)* the transformation of XSML models into GSL, focusing on the languages' characteristics, *(2)* a new composition mode in GCL for the semantic-preserving integration of XSML components, and *(3)* the application of our approach on a real-life distributed RIS subsystem under development.

II. Mapping XSML Components Into GSL

XSML is a textual statechart language reusing most elements of UML, e.g., it offers hierarchical states and transitions for describing state-based behavior and variables for expressing memory. However, as it is designed to describe critical functionalities, it aims for the easy interpretability of the models and discards UML elements for complex transitions, e.g., choice, merge, fork and join nodes, history states, as well

as entry and exit actions of states. Instead, it offers a powerful action language to capture the handling of variables as well as control flow (target states) in transitions. It also refines UML's operational semantics to *eliminate nondeterminism* in regard to transitions and orthogonal regions by introducing *priorities* and thus, a *sequential execution* of these elements.

Listing 1 presents an excerpt of one of the RIS components, called *ObjectHandler*, defined in XSML. The excerpt describes a state where the component waits for the confirmation message of a certain request. The waiting can end in an expected confirmation (entering *CmdConfirm*) or a fallback to a previous state (*WaitTS1Req*) via transitions due to an invalid message or a timeout while managing variables (potentially through functions) and dispatching messages via ports.

```
// Integer variables with different bitlengths
U8 SessionID;
U32 TimestampObj1, TimestampVk1;
...
state WaitTS2Req {
 // Timeout based on a parameter value
 timeout after (TimeConfirmationTimeout) {
  change WaitTS1Req;
 }
 // Transition triggered by a Rigel message
 event Rigel msg [msg.MsgType == MsgType.RigelMsgReqTs2]
     from PortIn {
  // Effects described in an action language
  if (SessionID != msg.SessionId)
   change CmdConfirm; // Potential target state
  else {
   var Rigel ansTs1Msg = ProcessReqTs2( // Function call
     TimestampVk1, TimestampObj1, SessionID);
   send ansTs1Msg to PortOut; // Message dispatch
   change WaitTS1Req; // Potential target state
  }
 }
}
```

Listing 1: Excerpt from the *ObjectHandler* XSML model.

The XSML-GSL model transformation must handle two characteristics of XSML besides the straightforward mapping of most model elements, e.g., regions, states and variables:

1) *Selecting target states* in transitions: In XSML, a single syntactic structure (if-else branches) is used to process an event, specifying not only the actions but also different target states (*change* statements) depending on the conditional branches; in contrast to GSL, where each transition shall have only a single, fixed target state.
2) *Introducing priorities*: In XSML, priorities of transitions and orthogonal regions are used to ensure deterministic behavior, which have to be represented in GSL.

Listing 2 illustrates how the transformation handles *target state selection* during event processing. First, a *transition* triggered by the corresponding event is introduced, which enters a *choice state*. This transition executes the corresponding actions while setting *auxiliary boolean variables* (*toWaitTS1Req* and *toCmdConfirm*) that identify the corresponding target states. From the choice state, a *set of transitions* is used, where each transition enters a proper target state depending on the values of the auxiliary variables referenced from their guard expressions. Note that this mapping retains the *atomicity* of transitions due to the semantics of choice states in GSL [5].

Introducing priorities to transitions and orthogonal regions is based on so-called *semantic variation points* offered by GSL, which – among others – support adjusting

- the *execution* of actions in orthogonal regions of composite states, which can be *sequential*, i.e., in the order of the declaration of regions, *unordered*, i.e, any region permutation is considered valid, and *parallel*, i.e., actions in orthogonal regions can interleave in any way; and,
- *priority* between enabled transitions leaving the same state – the *absence* of priority leads to *nondeterministic* choices between enabled transitions during execution.

With respect to the operational semantics of XSML, the transformation applies the *sequential* execution of orthogonal regions and transitions *prioritized* according to their *order of definition* ("earlier" defined transitions have a higher priority).

```
// Auxiliary boolean variables for target state selection
var toWaitTS1Req, toCmdConfirm : boolean
..
// Choice state for target state selection
choice WaitTS2Req_
..
// Selecting target states: single transition
transition from WaitTS2Req to WaitTS2Req_ when
     PortIn.msg [PortIn.msg::Value.MsgType ==
     MsgType::RigelMsgReqTs2] / {
 if (SessionID != PortIn.msg::Value.SessionId)
  toCmdConfirm := true; // Setting target state
 else {
  var ansTs1Msg : Rigel := ProcessReqTs2( // Function call
     TimestampVk1, TimestampObj1, SessionID);
  raise PortOut.message(ansTs1Msg);
  toWaitTS1Req := true; // Setting target state
 }
}
// Selecting target states: set of transitions
transition from WaitTS2Req_ to WaitTS1Req [toWaitTS1Req]
transition from WaitTS2Req_ to CmdConfirm [toCmdConfirm]
```

Listing 2: GSL elements derived from the transition triggered by a Rigel message in Listing 1.

III. Integrating XSML Components

Similarly to standalone statecharts, XSML also aims for a *deterministic* behavior at the level of component integration. Accordingly, it defines deterministic execution and communication semantics for integrated (composite) components that feature *(1)* the *sequential execution* of contained components and *(2)* their communication using immutable *messages* stored in *prioritized message queues*. Consequently, GCL must provide a composition mode that conforms to these characteristics and thus, we introduce the new *scheduled asynchronous-reactive* composition mode as previously introduced composition modes feature *parallel* execution with *message*-based communication (*asynchronous-reactive*) or *signal*-based communication (*cascade* and *synchronous-reactive*) [7].

The examined RIS subsystem represents the realization of the so-called Rigel protocol and comprises three components, namely *controlCenter*, *dispatcher* and *objectHandler*. Listing 3 describes the RIS subsystem model integrated in GCL using the *scheduled asynchronous-reactive* composition mode. The model has an integer parameter (*Timeout*) and two ports (*ControlPortIn*, *ControlPortOut*) for the transmission

of input and output messages defined in the *Rigel* interface. The model contains the above-mentioned standalone statechart components (the *objectHandler* also has a *Timeout* parameter) derived from XSML models whose ports are connected using *channels* to enable internal communication. Moreover, the control ports of the *controlCenter* component are *bound* to the external ports of the system.

The new composition mode supports a *cycle-based* execution mode in which components are executed *sequentially*. The execution order is defined in an execution list (*execute* keyword). A component can be referenced multiple times in the execution list, allowing its multiple execution in a single cycle. Regarding communication, the components interact using immutable *messages* stored in prioritized *message queues*. Such message queues can be defined using *asynchronous adapter* [7] models (*ControlCenter*, *Dispatcher* and *ObjectHandler*) that adapt statechart models to message-based communication. A message queue has the following fixed attributes: *(1)* stored message types, *(2)* priority, *(3)* capacity and *(4)* the handling of incoming messages in case the queue is full (discard the *incoming* or the *oldest* stored message). When selecting a message for processing, one is always retrieved from the highest priority non-empty queue.

```
scheduled-async RIS(Timeout : integer) [
 // System ports visible from the environment
 port ControlPortIn : requires Rigel
 port ControlPortOut : provides Rigel
] {
 // Contained components of the RIS
 component controlCenter : ControlCenter
 component dispatcher : Dispatcher
 component objectHandler : ObjectHandler(Timeout)
 // Binding the control center ports to the system ports
 bind ControlPortIn -> controlCenter.ControlPortIn
 bind ControlPortOut -> controlCenter.ControlPortOut
 // Channels for the inter-component communication
 channel [ controlCenter.PortOut ] -o)- [ dispatcher.
     ControlCenterPortIn ]
 channel [ dispatcher.ControlCenterPortOut ] -o)- [
     controlCenter.PortIn ]
 channel [ dispatcher.ObjectHandlerPortOut ] -o)- [
     objectHandler.DispatcherPortIn ]
 channel [ objectHandler.DispatcherPortOut ] -o)- [
     dispatcher.ObjectHandlerPortIn ]
 // Scheduling order of components
 execute controlCenter, dispatcher, objectHandler
}
```

Listing 3: RIS model integrated in GCL using the *scheduled asynchronous-reactive* composition mode.

IV. Formal Verification

The complete GCL model is mapped into low-level analysis models via a sequence of internal automated model transformations that take into account the composition mode. The analysis models can be verified with respect to manually defined properties using model checker back-ends integrated to Gamma. The results, i.e., whether the property holds in the model and potentially a diagnostic trace as proof, are automatically back-annotated to the source GCL model. Currently, UPPAAL, Theta and Spin are supported as back-ends, which are tailored to handling different models, e.g., UPPAAL supports timed behavior, Theta supports abstraction-based symbolic techniques, and Spin excels at checking parallel behavior. They also support different property specification languages, e.g., UPPAAL supports a restricted CTL, Theta supports reachability, and Spin supports LTL [11], providing a good portfolio for model checking.

In order to support the exhaustive verification of industrial-scale systems, the transformations feature several *model reduction* and *slicing* algorithms to reduce the state space of models under verification. The model reduction algorithms, which are independent of the verifiable properties and applied on the model in itself, reduce the following model elements:

- unused variables and input events with their parameters;
- unfireable transitions, e.g., due to the lack of triggering events or guards evaluating to constant false;
- unreachable states and regions without a functionality, e.g., with a single simple state without entry/exit actions.

Model slicing is conducted depending on the verifiable properties and reduce the following model elements:

- unreferenced enumeration literals;
- unreferenced variables and input events with their parameters that do not influence internal behavior.

V. Test Generation

Test generation based on the complete GCL model utilizes the verification functionalities presented in Sect. IV. As a general idea, in a testing context, a diagnostic trace for a GCL 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, we control model checkers in a way that they generate diagnostic traces (abstract test cases) to cover test targets specified as formal properties (*trap properties*) [12]. These abstract test cases then can be customized to different execution environments, e.g., Java and C.

Test targets can be specified based on the following model element based (structural), behavior- (interactional) and dataflow-based coverage criteria:

- *output event*, *state*, *transition* and *transition-pair* (pairs of transitions entering and leaving a certain state);
- *sending* (event raise) and *receiving/processing* (transition triggered by the event) of an event between two *communicating components*;
- *execution paths* between the definition (def) and the use/reading (use) of variables within standalone components and also between communicating components.

The generated tests can be used to detect faults in component implementations (e.g., missing implementation of transitions), interaction of components, and improper variable definitions and uses in system implementations.

Test targets are defined in terms of reachability properties, which are trivial only in the case of output events and states as these model elements can be directly referenced from properties. In other cases, the GCL model has to be *annotated* to enable describing the coverage of these criteria. Accordingly, transition and variable def-use coverage criteria

necessitate the injection of boolean variables indicating their coverage, whereas transition-pair and interaction coverage criteria require the injection of integer variables that store the ID of the covered elements. As the number of coverable elements can be large, the approach supports the *customization* of criteria, allowing the inclusion/exclusion of components and relevant model elements, e.g., states, transitions and ports.

In order to make test generation more efficient in terms of time and the size of the generated test set, the approach utilizes two optimization algorithms. After generating a *new abstract test case*, the first algorithm iterates through the still *uncovered* test targets and checks whether the test case also covers some of them [12]; such test targets get discarded. After covering *each test target*, the second algorithm is applied, which searches for test cases in the test set that are *prefixes* of other test cases [13]. Note that such test cases can exist even when the first algorithm is applied due to the nondeterministic order of processing test targets. Such test cases do not contribute to the coverage of additional criteria, and thus, can be discarded to further reduce the generated test set.

VI. Evaluation

We evaluated the feasibility of our test generation approach on the integrated RIS model, focusing on test generation *time* and the *size* of the generated test set *with test optimization* in the case of full *state*, *transition* and *interaction* coverage. The RIS model altogether has 22 regions, 38 states, 118 transitions, 10 variables and 13 clock variables. We set a 5ms timeout parameter value for the model and used UPPAAL as this back-end could manage the features of the RIS the most efficiently in terms of execution time. Table I contains the measurement results. We generated tests five times for each coverage criterion; the time-related values in the table are represented in seconds and refer to the median of these results (the test size related values do not change in different runs).

The results show that as the coverage criterion for testing gets finer, the number of test targets, generated tests and contained cycles increases; apart from one case concerning interaction coverage due to the large number of uncoverable interactions. In addition, the average generation time for a single test target also increases due to the complexity of injected annotations (auxiliary variables). Nevertheless, the results show the approach is feasible for industrial-scale models, as *every test target* for every criterion could be handled in less than 14 seconds on average without any complication, e.g., a timeout or out of memory error in the process.

	State	**Transition**	**Interaction**
#Test targets	38	118	387
#Generated tests	4	26	22
#Cycles in tests	30	230	240
$\Sigma\mathbf{T}$ (s)	243	950	5377
$\overline{\mathrm{T}}$ (s)	6.4	8.1	13.9

TABLE I: The number of *test targets*, *generated tests* and *cycles* in the generated tests, as well as the median joint *test generation time* and *average test generation* time for a *single test target* in *seconds* for full *state*, *transition* and *interaction* coverage in the integrated RIS model.

VII. Conclusion and Future Work

In this paper, we presented a formal verification and model-based integration test generation approach for distributed controllers, adapted to a real-life RIS subsystem. The adaptation necessitated the mapping of the XSML design language and its semantics into the internal languages of the Gamma framework and also the introduction of a new composition mode. Based on these extensions, automated formal verification and customizable test generation are supported for RIS design models. Our evaluation demonstrated the feasibility of the approach on an existing distributed RIS subsystem using different coverage criteria for test generation.

Subject to future work, we plan to extend the approach to support additional execution and communication modes during component integration, e.g., introduce shared *global message queues* for components, to aid engineers in experimenting with different integration semantics.

References

[1] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff, "Calm and Cadena: Metamodeling for component-based product-line development," *Computer*, vol. 39, no. 2, pp. 42–50, 2006.

[2] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, and E. Jackson, *OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems*. Berlin, H.: Springer, 2014, pp. 235–248.

[3] A. Ferrari, F. Mazzanti, D. Basile, and M. H. ter Beek, "Systematic evaluation and usability analysis of formal methods tools for railway signaling system design," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4675–4691, 2022.

[4] G. Lukács and T. Bartha, "Formal modeling and verification of the functionality of electronic urban railway control systems through a case study," *Urban Rail Transit*, vol. 8, 11 2022.

[5] B. Graics, "Documentation of the Gamma Statechart Composition Framework v0.9," Budapest Univ. of Technology and Economics, Tech. Rep., 2016, https://tinyurl.com/yeywrkd6.

[6] V. Molnár, B. Graics, A. Vörös, I. Majzik, and D. Varró, "The Gamma Statechart Composition Framework," in *40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden: ACM, 2018, p. 113–116.

[7] B. Graics, V. Molnár, A. Vörös, I. Majzik, and D. Varró, "Mixed-semantics composition of statecharts for the component-based design of reactive systems," *Software and Systems Modeling*, vol. 19, p. 1483–1517, 2020.

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

[9] A. Hartman and K. Nagin, "The AGEDIS tools for model based testing," *ACM Sigsoft Software Engineering Notes*, vol. 29, 07 2004.

[10] G. Hamon, L. de Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM)*, 01 2004, pp. 261–270.

[11] E. A. Emerson and J. Y. Halpern, ""Sometimes" and "not never" revisited: On branching versus linear time temporal logic," *J. ACM*, vol. 33, no. 1, p. 151–178, Jan. 1986.

[12] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

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