Modeling and Analysis of an Industrial Communication Protocol in the Gamma Framework

Bence Graics*†, István Majzik*
*Budapest University of Technology and Economics, Department of Measurement and Information Systems
Budapest, Hungary
†MTA-BME Lendület Cyber-physical Systems Research Group
Budapest, Hungary
Email: {graics, majzik}@mit.bme.hu

***Abstract*—Communication protocols are often designed on the basis of state-based models. During protocol design, the use of formal verification is indispensable, as concurrent behavior is notorious for hidden and sophisticated bugs. This paper presents a formal verification approach to verify an industrial communication protocol using the Gamma Statechart Composition Framework. Gamma is a modeling toolset for the design and analysis of reactive systems. It supports a family of modeling languages with formal semantics for the component-based definition of state-based behavior. It also supports formal verification by automatically mapping the defined models to the input formalisms of verification backends and back-annotating the results. The verification approach is presented in the context of the Orion industrial communication protocol. The verification approach supports the introduction of channel models with different message transmission characteristics and failure modes. Different execution modes of the components are also analyzed.**

I. Introduction

Communication protocols are inherently event-driven and are frequently designed using state-based models, e.g., statecharts [1]. Furthermore, communication protocols are often used in safety-critical systems where correct behavior is crucial, which makes formal modeling languages as well as sophisticated verification and validation (V&V) techniques, e.g., formal verification, necessary during the design process.

As communication protocols have multiple participants, the modeling language must support composition functionalities in addition to supporting individual component design. Also, to make formal verification feasible, the modeling language must have a formal semantics both at component and system level, defining how a standalone component is executed, and describing the execution and communication of contained components. Such a language can be supported by a modeling and analysis tool, which can facilitate the design and V&V of communication protocols.

The Gamma Statechart Composition Framework is such a tool, providing a language for composing individual statechart components (possibly created in other tools) while supporting verification and validation (V&V) capabilities. In this paper we propose a formal verification approach for communication protocols using the Gamma framework, which includes *1)* the construction of protocol participant models as well as channel models with different failure modes, *2)* the composition of protocol participant and channel models to form system models and *3)* model checking on the system models with automatic back-annotation of the results. The process is presented in the context of Orion, a master-slave communication protocol under design targeted to be used in the railway industry.

II. Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework [2] is an open-source, integrated modeling toolset to support the semantically sound composition of heterogeneous statechart components. The framework reuses statechart models of third-party tools and their code generators for separate components, e.g., Yakindu[1] and MagicDraw[2], thus UML/SysML state machine models are supported. The mapping of these external models to the internal statechart representation of Gamma (Gamma Statechart Language – GSL) is supported by automatic model transformations. The framework provides the Gamma Composition Language (GCL), which supports the interconnection of components according to different composition modes based on precise semantics. Furthermore, Gamma provides code generators for deriving implementation from defined models as well as test case generators for the analysis of component interactions. Gamma also supports system-level formal verification and validation (V&V) functionalities by mapping statechart and composition models into formal automata of the UPPAAL [3] model checker. Also, the automatic back-annotation of the verification results is supported.

GCL supports three composition modes, namely *synchronous*, *cascade* and *asynchronous*, which fundamentally determine the execution of the resulting composite models. The detailed introduction of these composition modes can be found in [4], here we include a summary of their properties.

Synchronous A synchronous model represents a coherent unit consisting of strongly coupled but concurrent components, which are executed in a lock-step fashion and communicate in a synchronous manner using signals.

Cascade Cascade models are special synchronous models whose components are executed in a sequential manner. Contained components can be considered as a set of filters applied sequentially to derive an output from an input.

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

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

III. Modeling of the Communication Protocol

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

A. Protocol Participants

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

- *Connect* and *Disconnect* events come from the environment and can be used as external commands to initiate a connection or break down an established connection. *Invalid* event is also an external event indicating an invalid status in the environment of the system.
- Events of the Orion protocol are transmitted between protocol participants and can be used to establish (*OrionConnReq*, *OrionConnResp* and *OrionConnConf*) or break down a connection (*OrionDisconnCause*), send data in established connections (*OrionAppData*) or keep established connection alive in the absence of transmittable data (*OrionKeepAlive*).

The initial state of the *master* statechart (depicted in Fig. 1) is *Closed*. Upon receiving a *Connect* event or after a specified timeout (*TReconn*: 5 seconds in the example), it goes to state *Connecting* while sending an *OrionConnReq* event to the slave. If it receives an *OrionConnResp* event within a specified time interval, it goes to state *Connected* while sending an *OrionConnConf* event to the slave. If in state *Connecting* it receives any other events, or does not receive any events in a specified time interval (*TConn*: 5 sec), it goes back to state *Closed* and sends an *OrionDisconn* event when necessary, that is, if the received event was not *OrionDisconnCause*. In state *Connected*, application specific data, or in the absence of data for a specified time interval (*TKeepAlive*: 4 sec) an *OrionKeepAlive* event are sent (child state *KeepAliveSendTimeout*). Also in state *Connected*, data as well as *OrionKeepAlive* events are received (child state *KeepAliveReceiveTimeout*). However, if any other event is received or no events are received in a specified time interval (*TInactive*: 5 sec), the master goes back to state *Closed* and sends an *OrionDisconn* event if necessary.

The *slave* statechart (see Fig. 2) is similar to the master.

The models can be automatically transformed to the GSL using the model transformers of Gamma, in which they can be validated based on statechart-related well-formedness rules [5]. According to the validators of Gamma, the presented statechart models are well-formed.

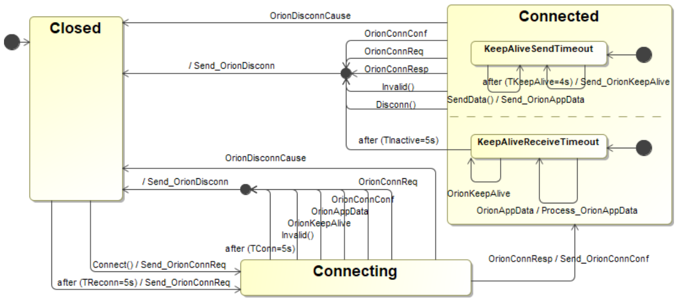

Fig. 1. The statechart model describing the behavior of the master component.

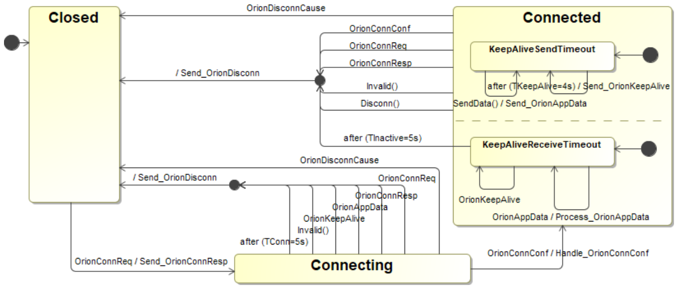

Fig. 2. The statechart model describing the behavior of the slave component.

B. Channel Models

Several failure modes of event transmission between protocol participants can be considered [6]. In this work we focus on *loss of events* and *delay of events* failure modes, as

- the *duplication of events* can be filtered using sequence numbering, this failure does not reach the protocol level,
- the *reordering of events* can be detected using sequence numbering, on protocol level this failure is mapped to the loss of these events, and
- the *alteration of event content* can be detected using integrity checking, on protocol level this failure is also mapped to the loss of these events.

Therefore, by focusing on *loss of events* and *delay of events*, we cover all relevant failure modes of [6].

In this work we defined five atomic channel models in Yakindu: one *ideal channel*, three models describing loss of events failure modes (*bursty message losing channel*, *arbitrary message losing channel* and *timed message losing channel*) and one model related to delay of events failure mode (*delay channel*). The following paragraphs present these models using graphical statechart representations. Note that these representations are simplified versions of the real models and include behavior only for a single event (*OrionConnReq*), however, additional events in the real models are handled analogously.

Fig. 3 depicts the *ideal channel* model. When it receives a certain event on its input, it forwards the event to its output, events are not lost or delayed.

Fig. 4 depicts the *bursty message losing channel* model, which models a channel that can lose a given amount (*LOST_MESSAGE_MAX*) of subsequent incoming events. It has two states, *Operating* (initial state) and *MessageLosing*. If

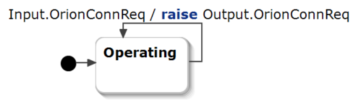

Fig. 3. The statechart model of the *ideal channel*.

the model receives a certain event in state *Operating* it either forwards the event to its output, or (if there has been no failure before) goes to state *MessageLosing* without forwarding the event. In state *MessageLosing*, the specified amount of events are absorbed before going back to state *Operating*. Note the nondeterministic nature of this model: the loss of subsequent events can start on any incoming event.

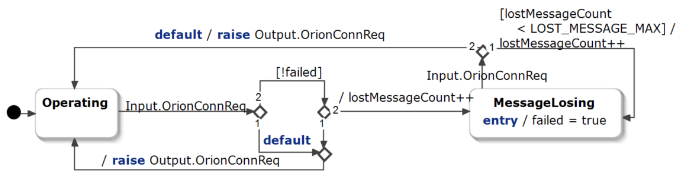
default / raise Output.OrionConnReq
[lostMessageCount < LOST_MESSAGE_MAX] / lostMessageCount++
Operating
Input.OrionConnReq
[!failed]
Input.OrionConnReq
/ lostMessageCount++
MessageLosing
entry / failed = true
default
/ raise Output.OrionConnReq

Fig. 4. The statechart model of the *bursty message losing channel*.

Fig. 5 depicts the *arbitrary message losing channel* model. It overapproximates the behavior of the *bursty message losing channel* model, as it supports the loss of events regardless of their order, that is, a given amount of events can be lost but these losses can occur any time, the lost events do not have to be necessarily subsequent.

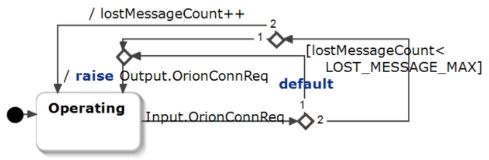
/ lostMessageCount++
[lostMessageCount< LOST_MESSAGE_MAX]
/ raise Output.OrionConnReq
default
Operating
Input.OrionConnReq

Fig. 5. The statechart model of the *arbitrary message losing channel*.

Fig. 6 depicts the *timed message losing channel* model, which loses messages that are received in a specified time interval. It has two states, *Operating* (initial state) and *MessageLosing*. In state *Operating*, incoming events are forwarded to the output. After a certain time (S sec), if the model has not failed before, it goes to state *MessageLosing*, where incoming events are absorbed. It goes back to state *Operating* after a specified time (E sec) and remains there.

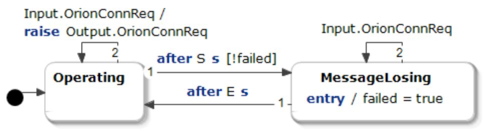

Fig. 6. The statechart model of the *timed message losing channel*.

Fig. 7 depicts the *delay channel* model, which delays the transmission of events with a given time. In this model each event type is handled in an orthogonal region. A region has two states, *Idle* (initial state) where there is no event in the channel, and *Forwarding* where the transmission of events of a certain type is delayed. If an event is received in state *Idle*, the model goes to state *Forwarding* where additional incoming events are queued (variable *messageCount*). After a specified time (T sec), the delayed event is forwarded. If there is no additional queued event, the model goes to state *Idle*, otherwise, it goes back to state *Forwarding*.

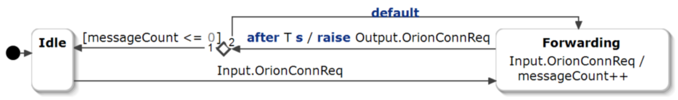
default
Idle
[messageCount <= 0]
after T s / raise Output.OrionConnReq
Forwarding
Input.OrionConnReq / messageCount++
Input.OrionConnReq

Fig. 7. The statechart model of the *delay channel*.

C. System Models

We analyzed the behavior of the Orion protocol considering different channel failure modes and different execution modes of the participants. Therefore, for each channel model we defined cascade, synchronous and asynchronous composite Gamma models, which differ only in the execution mode, the components and their connections are the same. In this work we focused on the time-driven behavior and the events of the Orion protocol in the master and slave components and did not consider the external events and commands that may directly close the connection.

Fig. 8 describes the GCL model the variations of which were used with different channel models and execution modes. It consists of a master component, a slave component, and two channel components that connect the output and input ports of the protocol participants. The concrete models differ only in the first keyword that can be either *sync*, *cascade* or *async*. All in all, fifteen composite system models were defined, five (as there are five channel models) for each composition mode. In the asynchronous composite models message queues with capacity 2 were used.

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

Fig. 8. The GCL model of protocol participants and channel models.

IV. Analysis of the Communication Protocol

We analyzed liveness properties of the system models introduced in Section III-C, that is, the reachability of system states using different channel models and execution modes. The analyzed properties (formalized in CTL) are the following.

- P_1 The system *can reach* a state in which both the master and the slave are in state *Connected*: `EF master.Connected && slave.Connected`.
- P_2 The system *must eventually reach* a state in which both the master and the slave are in state *Connected*: `AF master.Connected && slave.Connected`.

P_1 means the models do not contain fundamental faults. P_2, as a robustness property means the protocol is always able to recover despite the specified failure mode of the channel.

According to the verification executed in Gamma, P_1 *holds* in the case of every system model introduced in Section III-C. The analysis results for P_2 are shown in the following sections.

A. Synchronous Composition Mode

As the protocol has real-time timeouts, the fulfillment of the property depends on the execution frequency (indicated by f) of the system components in the case of each channel model.

In the case of the *ideal channel* model, f has to be higher than $4/\mathit{TConn}$ for the property to hold due to the *lock-step* execution mode of the components: event transmission between the master and the slave is delayed as events are transmitted through separate channel components (the value in the nominator refers to the number of components in the composite model). The timeout in state *Connecting* (both for the master and the slave) is *TConn* sec. Therefore, event *OrionConnResp* in response to *OrionConnReq* in the case of the master, and event *OrionConnConf* in response to *OrionConnReq* in the case of the slave have to be received in lesser time to enable the reaching of state *Connected*.

The property holds in cases of both the *bursty* and *arbitrary message losing channel* models for all *LOST_MESSAGE_MAX* values between 1 and 9, if f is higher than $4/\mathit{TConn}$.

In the case of the *timed message losing channel* model, the property was checked for the following S and E values: 4 and 9, 4 and 14, 4 and 19, 9 and 14, 9 and 19, 14 and 19 (so that multiples of parameters *TReconn* and *TConn* in the master and slave models fall into these intervals). If f is higher than $4/\mathit{TConn}$, the property holds.

In the case of the *delay message losing channel* model, f, the T parameter of the channel and the *TConn* parameter in the master and slave have to satisfy the following constraint: $2/f + T < \mathit{TConn}/2$. If this constraint is not satisfied, an execution of the components can exist where the master and slave get desynchronized due to the late arrival of messages and the *Connected* states are never reached at the same time.

B. Cascade Composition Mode

The cascade composition mode defines a *sequential* execution semantics. Similarly to the synchronous composition mode, the fulfillment of the analyzed property depends on the execution frequency of components. The property can be fulfilled in the case of every channel model, and in this composition mode the execution frequencies can be lower.

In the case of the *ideal channel*, *bursty-*, *arbitrary-* and *timed message losing channel* models, f has to be higher than $1/\mathit{TConn}$ for the property to hold, as components are executed in the following order: *master*, *m2S*, *slave*, *s2M*. Therefore, events from the master to the slave and from the slave to the master can be transmitted in a single execution cycle.

In the case of the *delay message losing channel* model, f, the T and the *TConn* parameters have to satisfy the following constraint: $1/f + T < \mathit{TConn}/2$.

C. Asynchronous Composition Mode

In the case of the asynchronous composition mode, there is no guarantee on the execution frequency of the components of the composite model. Therefore, in the case of any channel model, it is possible to delay the execution of either the master or slave component in state *Connecting* until the timeout is reached (*TConn* sec). Thus, the property *does not hold* regardless of the defined channel models. To examine this problem, we introduced constraints on the execution frequency of the components in the formal models. We observed that if the execution frequencies of the components are in the order $f(\mathit{master}) < f(\mathit{m2S}) < f(\mathit{slave}) < f(\mathit{s2M})$, and are higher than $4/\mathit{TConn}$ in the cases of the first four channel models, or the $2/f + T < \mathit{TConn}/2$ constraint holds in the case of the *delay* channel model, the property holds.

V. Conclusion

We proposed a verification approach for communication protocols using Gamma in the context of Orion, an industrial master-slave communication protocol. We defined multiple channel models describing loss of events and delay of events failure modes. Using model checking, we verified whether the system *1)* might eventually and *2)* must eventually reach a state in which both the master and slave are in state *Connected*.

In the future we plan to investigate the composition of channel models in a single GCL component where the operating channel model is selected by a selector component. This way, both the model construction and the analysis phases could be simplified as the number of resulting system models would decrease to three. We also aim to introduce full support for execution frequency constraints in asynchronous components.

Acknowledgment Supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013) and the ÚNKP-19-3 New National Excellence Program of the Ministry for Innovation and Technology.

References

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

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

[3] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," 2006.

[4] B. Graics and V. Molnár, "Mix-and-match composition in the Gamma Framework," in *25th Minisymposium, Department of Measurement and Information Systems*, Budapest, Hungary, January 2018.

[5] B. Graics, "Documentation of the Gamma Statechart Composition Framework," Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems, Tech. Rep., 2016. [Online]. Available: https://inf.mit.bme.hu/en/gamma/

[6] S. Procter and P. Feiler, "The AADL error library: An operationalized taxonomy of system errors," *Ada Lett.*, vol. 39, no. 1, p. 6370, Jan. 2020. [Online]. Available: https://doi.org/10.1145/3379106.3379113