Generating Scenarios by Multi-Object Checking

Maik Kollmann[1]

Institute of Information Systems
Technical University Braunschweig
Mühlenpfordtstraße 23, 38106 Braunschweig, Germany

Yuen Man Hon[2]

Institute for Railway Systems Engineering and Traffic Safety
Technical University Braunschweig
Pockelsstraße 3, 38106 Braunschweig, Germany

Abstract

These days, many systems are developed applying various UML notations to represent the structure and behavior of (technical) systems. In addition, for safety critical systems like Railway Interlocking Systems (RIS) the fulfillment of safety requirements is demanded. UML-based Railway Interlocking (UML-based RI) is proposed as a methodology in designing and developing RIS. It consists of infrastructure objects and UML is used to model the system behavior. This design is validated and demonstrated by using simulation with Rhapsody. Automated verification techniques like model checking have become a standard for proving the correctness of state-based systems. Unfortunately, one major problem of model checking is the state space explosion if too many objects have to be taken into account. Multi-object checking circumvents the state space explosion by checking one object at a time. We present an approach to enhance multi-object checking by generating counterexamples in a sequence diagram fashion providing scenarios for model-based testing.

Keywords: Scenario Generation, Counterexample Generation, State Modeling, Multi-Object Checking, UML-based Railway Interlocking System

1 Introduction

Nowadays, during different phases of system development, Unified Modeling Language (UML) notations [19] is used as the modeling tool to specify the structure and behavior of (technical) systems. These systems contain components or objects that interact via channels or a bus in order to provide a specified functionality. In

---

[1] Email: M.Kollmann@tu-bs.de
[2] Email: Y.Hon@tu-bs.de

addition, for safety critical systems like Railway Interlocking Systems (RIS) the fulfillment of safety requirements is demanded. The proof that a safety critical system provides safe operation determines the conformance to the desired safety integrity level (SIL) of such a system [5,2]. In general, safety critical systems are demanded to reach SIL 4, the highest level. According to EN 50128 [5], SIL 4 RIS highly recommend a proof that the safety requirements are fulfilled up to a tolerable level by the RIS software.

RIS are responsible for establishing safe routes for trains that are scheduled to pass through or stop at a railway station. Safe routes ensure that trains cannot be driven into tracks that are occupied or may become occupied by other trains. A safe route ensures the proper settings of infrastructure elements like points and signals along the route. These elements can be modeled using UML class diagrams and UML state machines [15,3]. The next step is to prove that a model of a concrete interlocking is working correctly and that it conforms the safety requirements. Such requirements are the absence of conflicting routes, etc. Automated verification techniques like model checking have become a standard for proving the correctness of state-based systems. Unfortunately, model checking suffers from the state space explosion problem if too many objects have to be taken into account.

Multi-object checking [11] circumvents the state space explosion by its nature checking one object at a time. Multi-object checking relies on the sound and complete multi-object logics D_1 and D_0 [9] that are based on well-known logics like CTL or LTL. D_1 is supposed to be more intuitive for specification, as the interaction among different objects can be specified. However, it cannot be verified directly applying model checking. Formulas in D_1 can be translated into an equivalent set of D_0 formulas. Each checking condition is bound to a particular object in a D_0 formula. As a result, D_0 formulas can be verified automatically by model checking.

There are similar properties between multi-object checking and traditional model checking. They provide an automatic proof that a certain (safety) property holds. In addition, model checking provides the opportunity to analyze a system model if the checking condition does not hold. If a model violates a checking condition, multi-object checking provides a counterexample from a model checking tool for the outermost object. Taking the mentioned communication with other objects into account, this counterexample trace had to be analyzed manually. We propose an observer-based approach to synthesize a multi-object counterexample that is closed with respect to interaction of objects.

Model checking tools like SPIN [14] or SMV/NuSMV [18,4] incorporate the ability to illustrate that a model does not satisfy a checking condition with a textual, a tabular or a sequence chart-like representation of the involved states. We believe that a graphical representation like message sequence charts is the most convenient one from a users point of view to illustrate a counterexample in a multi-object system. Beyond providing a graphical representation for certain scenarios, sequence charts have been successfully applied to generate test cases [7]. In this contribution, we present an approach for generating test cases automatically based on the system specification instead of deriving them manually.

2 Railway Interlocking Systems

RIS are often designed specifically according to the layouts of stations. When the layout of a station is changed, the corresponding RIS has to be modified. This causes high costs for resources and time [23]. In order to reduce the effort in modifying RIS, one can develop RIS under the object oriented approach. With this approach, it is only necessary to specify the new neighboring structure of the objects when the layout of the station is changed instead of modifying the whole interlocking system completely.

The geographical interlocking is used as the internal logic of the interlocking system that consists of infrastructure elements as objects. The objects interact among each other with events and actions to develop safe routes. Objects, events and actions can be captured by using state modeling. In [1] Statemate [13] is proposed and applied to model the functional requirements of interlocking systems. In contrast, in this work, UML state machines are used for state modeling. RIS that are developed by applying the geographical interlocking and using UML as specification tool are called UML-based Railway Interlockings (UML-based RI).

Figure 1 shows the railway station CStadt that is used to check the feasibility of developing a UML-based RI. There are different kinds of infrastructure elements that are located within this station: tracks, points and signals.
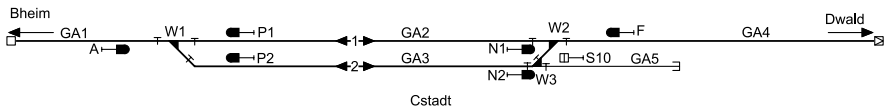

Fig. 1. Track layout of station CStadt

Tracks are the basic infrastructure elements that trains move on. The authorized movement of railway vehicles on the tracks can be classified into train movements and shunting movements. Train movements are only allowed on main tracks, while making up a train, called shunting movements, are undergone only on sidings [20]. In figure 2, main tracks are $GA1$, $GA2$, $GA3$ and $GA4$. Only train movements are analyzed in the current model of UML-based RI. Instead of moving only straight ahead, trains can also turn to another track via turnouts or crossing. Turnouts and crossing are composed of points. Points (e.g., $W1$) can be set to right and left position. In order to ensure the train can be driven into the current track, RIS need to ensure the correct position of points.
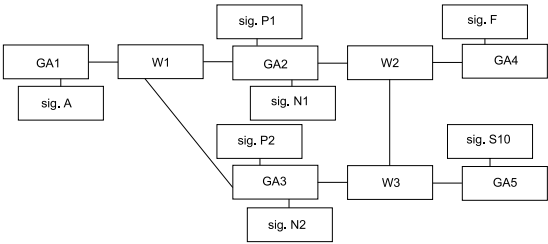

Fig. 2. Station CStadt in UML-based RI

Along the track, one can find another kind of infrastructure component: the signal. Signals (e.g., $P1$) can be used to control train movements by showing the aspects. Two types of aspects will be considered in this work, they are clear and stop aspect. Some signals can also indicate the allowed speed of the corresponding track section. Indication of speed will not be considered in the current work.

As mentioned before, the main task of RIS is to develop a safe route for the approaching train. There are two requirements of a safe route. First, the infrastructure elements along the route have been properly set. For example, the points are set in the correct positions and no other trains can be driven into the safe route from the divergent direction. Flank protection is used to prevent other trains to drive into a safe route via points. Furthermore, no conflicting routes are issued at the same time as the safe route. This means the infrastructure elements which belong to the route can only be used exclusively by one train. If the requested route of the approaching train fulfills those requirements, in other words, it is a safe route, then this route and the infrastructure elements along this route will be locked exclusively for this train. No other train can use the same route, so that collisions can be avoided. The mentioned safety requirements can be ensured by executing two procedures: checking the availability of infrastructure elements and providing flank protection for the route.

3 Multi-Object Systems and Multi-Object Checking

One of the possible solutions in handling the state space explosion is to develop a method, such that a condition can be verified without building up the complete state space and only those state variables of objects that are involved in the condition are considered during the verification. In this approach, the correctness of a formula which involves a single object can be verified by checking the state space of this object locally and this type of formula can be called a local condition. When there are more objects defined, a formula is called a global condition. This global condition can be broken down into a set of local conditions of each object and communications between objects. The correctness of the global condition can be ensured by checking those local conditions and the existence of communications among the objects. Multi-object checking is a method that comprises the above ideas. It can be used to verify state-based multi-object system efficiently without building the complete state space. In multi-object checking, the system consists of objects and objects communicate among each other synchronously in an RPC-like fashion. The communication between objects must be specified before the verification. Each of the objects has a signature that describes its properties, for example, its attributes, actions and valid propositions. This signature can for example be captured in an Finite State Machine (FSM).

Let I be a finite set of identities representing sequential objects. Misusing terminology slightly, we speak of *object i* when we mean the object with identity i. Each object $i \in I$ has an individual set P_i of atomic state predicate symbols, its signature. In applications, these predicate symbols may express the values of attributes, which

$$
\begin{aligned}
D_1^i \;&::=\; i.L_1^{i_{LTL}} \\
F_1^{i_{LTL}} \;&::=\; i{:}L_1^{i_{LTL}} \mid i!L_1^{i_{LTL}} \mid i?L_1^{i_{LTL}} \mid \\
&\qquad i{:}_{last}L_1^{i_{LTL}} \mid i!_{last}L_1^{i_{LTL}} \mid i?_{last}L_1^{i_{LTL}} \mid \\
&\qquad i{:}_{untilnow}L_1^{i_{LTL}} \mid i!_{untilnow}L_1^{i_{LTL}} \mid i?_{untilnow}L_1^{i_{LTL}} \mid \\
L_1^{i_{LTL}} \;&::=\; \bot \mid \mathcal{P}_i \mid \neg L_1^{i_{LTL}} \mid L_1^{i_{LTL}} \wedge L_1^{i_{LTL}} \mid L_1^{i_{LTL}} \vee L_1^{i_{LTL}} \mid \\
&\qquad L_1^{i_{LTL}} \Rightarrow L_1^{i_{LTL}} \mid L_1^{i_{LTL}} \Leftrightarrow L_1^{i_{LTL}} \mid \\
&\qquad \mathsf{X}\,L_1^{i_{LTL}} \mid \mathsf{F}\,L_1^{i_{LTL}} \mid \mathsf{G}\,L_1^{i_{LTL}} \mid L_1^{i_{LTL}}\,\mathsf{U}\,L_1^{i_{LTL}} \mid \\
&\qquad \mathsf{Y}\,L_1^{i_{LTL}} \mid \mathsf{O}\,L_1^{i_{LTL}} \mid \mathsf{H}\,L_1^{i_{LTL}} \mid L_1^{i_{LTL}}\,\mathsf{S}\,L_1^{i_{LTL}} \mid \\
&\qquad @{:}J \mid @!J \mid @?J \mid C_1^{i_{LTL}} \\
C_1^{i_{LTL}} \;&::=\; \dots \mid F_1^{j_{LTL}} \mid \dots \qquad (j \in J \,,\; J \equiv I \setminus \{i\})
\end{aligned}
$$

Fig. 3. Syntax definition of Multi-Object Logic D_1 (with local logic LTL)

actions are enabled, which actions have occurred, etc.

We use the multi-object logic D_1 (cf. figure 3) as described in [9,10] with a local logic $i.L_1^{i_{LTL}}$ over signature P_i for each object $i \in I$. D_1 allows to use formulas $j{:}L_1^{j_{LTL}}$, $j!L_1^{j_{LTL}}$, $j?L_1^{j_{LTL}}$ etc. for any other object $j \in I$ as subformulas within $i.L_1^{i_{LTL}}$. These constituents are called communication subformulas.

The global conditions that involve more than one object are specified by the multi-object logic D_1. Logic D_1 allows one to specify temporal formulas or propositions that an object i fulfills locally. It also allows one to specify temporal formula or propositions that other objects satisfy for an object if there is synchronized communication between them. For example, a formula $o_1.(\neg\mathsf{EF}(state = critical \wedge o_2{:}(state = critical))$) means that there exists a synchronized communication currently between objects o_1 and o_2, such that o_2 guarantees for o_1 that o_2 is in the critical state (cf. figure 4). $o_2{:}(state = critical)$ is called the communication sub-formula in this method.
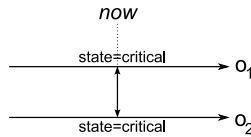
Fig. 4.   Objects o_1 and o_2 must not reside in the critical section together: $o_1.(\neg\mathsf{EF}(state = critical \wedge o_2{:}(state = critical)))$

There is a sublogic of D_1 called D_0 (cf. figure 5). Formulas that involve communication with different objects cannot be expressed in D_0 directly. However, the synchronized communication between two objects can be explicitly specified with the properties of the objects in D_0.

[9] presents a transformation to break global D_1 checking conditions down into sets of D_0 conditions and communication symbols (cf. figure 6). These symbols have to be matched with existing ones according to the communication requirements. In addition, D_0 conditions can be verified locally. Informally, the communication

$$
\begin{aligned}
D_0^i \ &::= \ i.L_0^{i_{LTL}} \mid i.C_0^i \\
L_0^{i_{LTL}} \ &::= \ \bot \mid \mathcal{P}_i \mid \neg L_0^{i_{LTL}} \mid L_0^{i_{LTL}} \wedge L_0^{i_{LTL}} \mid L_0^{i_{LTL}} \vee L_0^{i_{LTL}} \mid \\
&\quad L_0^{i_{LTL}} \Rightarrow L_0^{i_{LTL}} \mid L_0^{i_{LTL}} \Leftrightarrow L_0^{i_{LTL}} \mid \\
&\quad \mathsf{X}\, L_0^{i_{LTL}} \mid \mathsf{F}\, L_0^{i_{LTL}} \mid \mathsf{G}\, L_0^{i_{LTL}} \mid L_0^{i_{LTL}}\, \mathsf{U}\, L_0^{i_{LTL}} \mid \\
&\quad \mathsf{Y}\, L_0^{i_{LTL}} \mid \mathsf{O}\, L_0^{i_{LTL}} \mid \mathsf{H}\, L_0^{i_{LTL}} \mid L_0^{i_{LTL}}\, \mathsf{S}\, L_0^{i_{LTL}} \mid \\
&\quad @{:}J \mid @!J \mid @?J \\
C_0^i \ &::= \ \dots \mid (\mathcal{P}_i \Rightarrow j.\mathcal{P}_j) \mid \dots \qquad\qquad (j \in J \,,\; J \equiv I \setminus \{i\})
\end{aligned}
$$

Fig. 5. Syntax definition of Multi-Object Logic D_0 (with local logic LTL)

symbols are determined inside out. A D_1 formula ψ holds iff the outermost D_0 formula ψ' holds. This is elaborated in [17,11].

$$
\psi \ \equiv \ i.(\dots \ j{:}(\varphi) \ \dots)
$$
$$
\psi' \ \equiv \ i.(\dots \ q{:}j \ \dots) \qquad\qquad \varphi' \ \equiv \ j.(q{:}i \Rightarrow \varphi)
$$

Fig. 6. D_1 to D_0 transformation

3.1 Model Checking

Multi-object checking has been demonstrated successfully using model checking to determine the communication symbols and to match with existing ones according to the communication requirements in [11]. Model checking consists in verifying a system against a checking condition, also called a formula ($M \models \phi$, M is the model and ϕ is the formula).

The system that needs to be verified is specified as a (labeled) transition system T and is called the model M. A transition system $T = (S, s_0, L, \rightarrow)$ consists of a set of states S ($s_0 \in S$ is the initial state) with labeled transitions (L set of labels; $\rightarrow \ \subseteq S \times L \times S$ is a set of labeled transitions) [24].

Any checking condition that the system needs to satisfy is defined in temporal logic [16,6] and is called formula ϕ. In model checking, temporal logic is used to describe the properties of the system over time. Temporal logic models time as a sequence of states. As a result, one can use temporal logic to describe conditions that have to hold as the system evolves [16]. Unfortunately, model checking suffers from the state space explosion problem [22] if too many objects have to be taken into account during the process of verification.

A widely used feature of model checking is the ability to generate counterexamples if a checking condition is evaluated to FALSE. Each counterexample consists of a sequence of states illustrating the inconformity. The quality of counterexamples has been discussed in [12].

As no model checking tool can determine which sequence of states is suitable to

illustrate the error if different sequences may be chosen, estimating the quality of a specific counterexample is based on the view of the user. However, little progress has been achieved in this area [8]. The representation of counterexamples varies among the model checking tools. Most of them offer a textual representation of the sequence of states whereas other tools make use of tables or sequence diagrams.

3.2 Multi-Object Checking Counterexamples

As mentioned above, in multi-object checking, a D_1 formula holds iff the outermost D_0 formula holds. Otherwise, a counterexample has to be generated if this D_0 formula does not hold. If model checking is used to verify the outermost D_0 formula, an initial part of a counterexample is generated. This forms a part of a global counterexample and it is called a local counterexample. If the initial local counterexample does not contain communication with further objects, it is the global multi-object checking counterexample. Similarly, whenever the local counterexample contains communication with further objects, the local counterexample is a part of a global counterexample.

We concentrate on the class of multi-object checking counterexamples in which communication among objects exists. The initial state of all transition systems, does not contain any communication. Consequently, the initial local counterexample refers to model checking counterexamples of finite or infinite length. The order of communication between the objects's communication partners has to be preserved. The algorithm is executed as follows:

- Apply model checking for each object which is identified as a communication partner of the object under investigation.

- Check whether the communication can be preserved for each of the objects.

- Whenever a further communication partner is discovered iterate the algorithm until all communication is covered.

This algorithm works on the grounds that each communication partner provides a corresponding communication scheme. As it has been mentioned before, it is difficult to determine the expressiveness of a counterexample, the selection of a random counterexample for each object does not guarantee that a global counterexample can be generated. Whenever a communication partner of an object does not provide a demanded communication scheme one of the possibilities is that, the counterexample of this object has not been properly chosen. This issue can be solved by automatically selecting a different counterexample for the initiating object that shows a different communication scheme. Alternatively, users can interactively examine the part of the global counterexample that has been generated so far.

Both solutions show drawbacks. The first solution implements a kind of greedy/backtracking algorithm with all the well-known (dis-)advantages. The more often local counterexamples have to be regenerated, the less efficient the algorithm is. Finally, the global counterexample may not illustrate the error that clear as expected. In contrast, applying the second strategy, the user may not have enough information to decide whether the partial global counterexample shows the origin

of the error. These observations constitute the following strategy for verification driven counterexample generation:

- Depending on a user-specified timeout, generate different local counterexamples.

- If time runs out, the user can judge whether the actual counterexample is useful.

- If the user cannot take any advantage of the given counterexample, he can guide the algorithm by providing domain knowledge.

3.3   Application of the Counterexample Generation Algorithm to UML-based RI

We have introduced a UML-based RI in Section 2. Checking conditions (e.g., if an infrastructure object o is not in use ($state = unused$), it is always possible to establish a route ($lock = route$) using this object: $F := o.(\mathsf{G}\ (state = unused) \Rightarrow \mathsf{F}\ (lock = route))$ that the model is expected to fulfill need to be defined. In [15], similar requirements have been checked by simulation, model checking and multi-object checking.

The algorithm of generating counterexamples that has been mentioned can be demonstrated by checking condition F. As checking condition F does not hold initially, we have easily discovered the erroneous situation: $GA3$ may not be in use but it may receive a route request from both neighbors $W3$ and $W1$. In this case the safe (but not functional) response is to reject both route requests rr. We have applied the counterexample generation algorithm and created two observers for objects $W3$ and $W1$ in the first step (cf. figure 7).
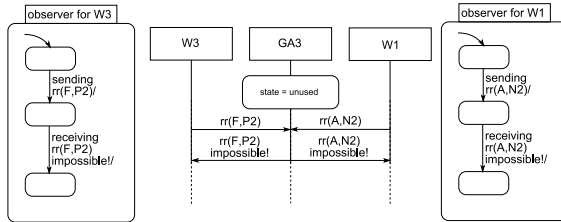

Fig. 7. Rejection of two concurrent route requests

The observer for $W3$ monitors the behavior of $W3$ such that a transition $t' \in T'$ in the observer automaton is a projection of those transitions $t \in T$ in $W3$ that have the same communication. A state in the observer automaton represents states and transitions that occur between the transitions t_i and t_{i+1} in the automaton of $W3$. Checking whether $W3$ can communicate with $GA3$ as expected can be evaluated by checking if the behavior of the observer automaton is possible (cf. figure 8). $W3$ and its observer synchronize on the transitions specified in the observer.

The generated sequence chart in figure 9 presents a global scenario out of the interlocking's perspective. It has been generated in the same way as described above.
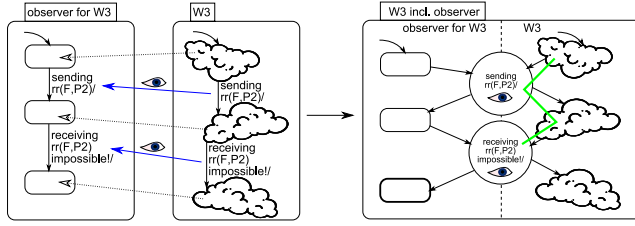
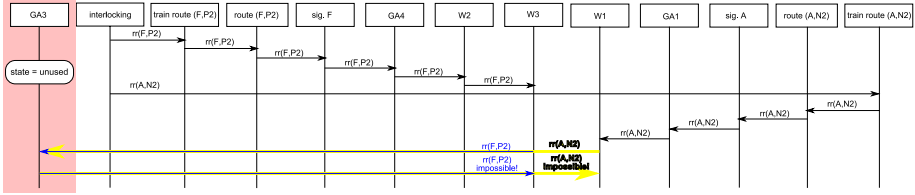Fig. 8. Creation of two observers for objects $W3$ and $W1$

Fig. 9. Conflicting routes may be established one after the other

3.4   Generation of Test Cases by Multi-Object Checking

Test cases are typically derived manually from a given specification. This derivation models a part of the system. These test cases can also be used to model the developed system's behavior. Both, the system design model and the test model may contain errors.

In the railway domain, many checking conditions ϕ, given by a (formal) specification, define implications that some conclusion ω holds under some premises λ [21]. Most of the premises and conclusions define a certain state in the system and constitute checking conditions which share the following structure: $\mathsf{G}((\lambda_1 \wedge \lambda_2 \wedge \ldots \lambda_n) \Rightarrow \mathsf{F}\omega)$. As each $\lambda_x, 1 \leq x \leq n$ belongs to a particular object i, a D_1 formula that is compliant to the communication requirements among objects can be formulated. A formula $\phi^{example} := \mathsf{G}((\lambda_{shared\ track\ element} \wedge \lambda_{route_1} \wedge \lambda_{route_2}) \Rightarrow \mathsf{F}(\omega_{shared\ track\ element}))$ is translated as follows:

$$\phi^{\mathsf{D}_1\ example} := \ shared\ track\ element.(\mathsf{G}((\lambda_{shared\ track\ element} \wedge \\ route_1{:}(\lambda_{route_1}) \wedge route_2{:}(\lambda_{route_2})) \Rightarrow \mathsf{F}(\omega_{shared\ track\ element})))$$

Let all such checking conditions ϕ evaluate to TRUE by verifying the system model beforehand. We automatically derive checking conditions ϕ_{test} by negating the conclusion: $\mathsf{G}(\lambda_1 \wedge \lambda_2 \wedge \ldots \lambda_n \Rightarrow \neg\mathsf{F}\omega)$. The negation is applied to the D_1 formulas as well. We derive the following checking condition to generate a test case for checking condition $\phi^{\mathsf{D}_1\ example}$:

$$\phi_{test}^{\mathsf{D}_1\ example} := \ shared\ track\ element.(\mathsf{G}((\lambda_{shared\ track\ element} \wedge \\ route_1{:}(\lambda_{route_1}) \wedge route_2{:}(\lambda_{route_2})) \Rightarrow \\ \neg\mathsf{F}(\omega_{shared\ track\ element})))$$

Such checking conditions obviously evaluate to FALSE by multi-object checking

and suitable counterexamples are generated applying the greedy/backtracking algorithm (cf. Section 3.2). They illustrate possible evolutions of the system from the initial state to the state in which the condition ($\neg F\omega$) does not hold. These evolutions can be used to specify test cases automatically if a mapping of states in the design model to the implementation is given.

In [7], a translation of Message Sequence Charts into *Testing and Test Control Notation Version 3* (TTCN-3) test cases is given. In combination with the generated sequence chart for a checking condition $\phi_{test}^{D_1}$ and an appropriate mapping to the implementation, a TTCN-3 test case can be derived easily.

For the railway domain, requirements coverage has been accepted as the coverage criterion with regard to all requirements that apply to already installed interlocking systems, as the risk generated by such a system has been tolerated for years or even decades. If all requirements are atomic in such sense that each of the requirements belongs to a single test case, applying all these test cases successfully will finalize the test process.

4   Conclusion

In this contribution, a strategy for generating counterexamples for multi-object checking is described. We have demonstrated the usefulness of our strategy by a case study featuring a UML-based RI. RIS are considered as safety critical systems. The guarantee of the correct behavior throughout the system lifecycle is demanded. In order to save resources in developing and modifying RIS for amended railway layouts, an object oriented approach for establishing interlocking systems is investigated in this work. Infrastructure elements are considered as infrastructure objects in a UML-based RI. The objects of a RIS cooperate with each other to function as an interlocking system.

Multi-object checking has been successfully applied to verify UML-based RIS. We concentrate on verifying the safety aspect of the model. The provided graphical counterexample in this contribution helped correcting the state machines displaying the obviously unhandled situation. We believe that this methodology that improves the understanding and communication among professions of different disciplines, can improve the whole development process of a system.

We have also shown a further step of improving system development by designing a more comprehensible verification strategy. It provides illustrative counterexamples and generates test cases automatically. Therefore, we have demonstrated how TTCN-3 test cases can be derived from checking conditions during the early stage of system development.

The positive feedback from railway engineers who are familiar with sequence chart notations suggests the further development of the user interface of our tool. Future work will enhance the counterexample generation process by the concurrent determination of multi-object checking communication symbols and reusing interim results during the verification process.

References

[1] Banci, M., A. Fantechi and S. Gnesi, *The role of format methods in developing a distributed railway interlocking system*, in: E. Schnieder and G. Tarnai, editors, *Proc. of the 5th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)* (2004), pp. 220–230.

[2] Bell, R., *Introduction to IEC 61508*, in: T. Cant, editor, *Tenth Australian Workshop on Safety Critical Systems and Software (SCS 2005)*, CRPIT **55** (2005), pp. 3–12.

[3] Berkenkötter, K. and U. Hannemann, *Modeling the Railway Control Domain Rigorously with a UML 2.0 Profile.*, in: J. Górski, editor, *Computer Safety, Reliability, and Security, 25th International Conference, SAFECOMP 2006, Gdansk, Poland, September 27-29, 2006, Proceedings*, 2006, pp. 398–411.

[4] Cavada, R., A. Cimatti, M. Benedetti, E. Olivetti, M. Pistore, M. Roveri and R. Sebastiani, *NuSMV: a new symbolic model checker*, http://nusmv.itc.it/ (2002).

[5] CENELEC, "EN 5012{6|8|9} – Railway Applications Dependability for Guided Transport Systems; Software for Railway Control and Protection Systems; Safety-related electronic railway control and protection systems," (1994-2006).

[6] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," MIT Press, 2000.

[7] Ebner, M., *TTCN-3 Test Case Generation from Message Sequence Charts*, Technical Report IFI-TB-2005-02, Institut für Informatik, Georg-August-Universität Göttingen, Germany (2005), presented at ISSRE04 Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL), IRISA, Rennes, France, 2. November 2004.

[8] Edelkamp, S., A. L. Lafuente and S. Leue, *Directed explicit model checking with HSF–SPIN*, Lecture Notes in Computer Science **2057** (2001), pp. 57+.

[9] Ehrich, H.-D. and C. Caleiro, *Specifying communication in distributed information systems*, Acta Informatica **36** (2000), pp. 591–616.

[10] Ehrich, H.-D., C. Caleiro, A. Sernadas and G. Denker, *Logics for Specifying Concurrent Information Systems*, in: J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, Kluwer Academic Publishers, 1998 pp. 167–198.

[11] Ehrich, H.-D., M. Kollmann and R. Pinger, *Checking Object System Designs Incrementally*, Journal of Universal Computer Science **9** (2003), pp. 106–119.

[12] Groce, A. and W. Visser, *What went wrong: Explaining counterexamples*, in: *SPIN Workshop on Model Checking of Software*, 2003, pp. 121–135.

[13] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot, *Statemate: a working environment for the development of complex reactive systems* (2002), pp. 135–146.

[14] Holzmann, G. J., *The model checker SPIN*, Software Engineering **23** (1997), pp. 279–295.

[15] Hon, Y. M. and M. Kollmann, *Simulation and Verification of UML-based Railway Interlockings*, in: S. Merz and T. Nipkow, editors, *Preliminary Proceedings of the 6th International Workshop on Automated Verification of Critical Systems*, Nancy, France, 2006, pp. 168–172.

[16] Huth, M. R. A. and M. D. Ryan, "Logic in Computer Science - Modelling and reasoning about systems," Cambridge University Press, 2000.

[17] Kollmann, M. and Y. M. Hon, *Generation of Counterexamples for Multi-Object Systems*, in: E. Schnieder and G. Tarnai, editors, *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)* (2007), to appear.

[18] McMillan, K. L., "Symbolic Model Checking," Kluwer Academic Publishers, 1996, second edition.

[19] OMG, "UML 2.0 Superstructure and Infrastructure," (2005).

[20] Pachl, J., "Railway Operation and Control," VTD Rail Publishing, New Jersey, 2004.

[21] Pavlovic, O., R. Pinger, M. Kollmann and H. Ehrich, *Principles of Formal Verification of Interlocking Software*, in: E. Schnieder and G. Tarnai, editors, *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)* (2007), to appear.

[22] Valmari, A., *The state explosion problem*, in: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets* (1998), pp. 429–528.

[23] van Dijk, F., W. Fokking, G. Kolk, P. van de Ven and B. van Vlijmen, *Euris, a specification method for distributed interlockings*, in: *SAFECOMP '98: Proceedings of the 17th International Conference on Computer Safety, Reliability and Security* (1998), pp. 296–305.

[24] Winskel, G. and M. Nielsen, *Models for concurrency*, in: S. Abramsky, D. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, Oxford University Press, 1995 .