# Runtime Safety Analysis for Safe Reconfiguration

Claudia Priesterjahn, Christian Heinzemann, Wilhelm Schäfer
Heinz Nixdorf Institute and Department of Computer Science, Software Engineering Group
University of Paderborn
Paderborn, Germany
[cpr|c.heinzemann|wilhelm]@upb.de

Matthias Tichy
Software Engineering Division,
Chalmers University of Technology and University of Gothenborg
Gothenborg, Sweden
tichy@chalmers.se

*Abstract*—**Modern technical systems are increasingly built to exhibit self-x properties as, e.g., self-healing or self-optimization. For this, they require adaptation at runtime. This is even true for embedded or mechatronic systems which often operate in safety-critical environments. There, the effects of the adaptation with respect to safety must be analyzed carefully. However, not all parameters needed for safety analyses, e.g., the concrete system architecture, are known at design time. Consequently, safety analyses need to be executed during runtime. Current approaches of runtime safety analysis typically react to anomalies that already occurred in the system. Thus, unsafe system states cannot be excluded completely. We present a runtime safety analysis that prevents system states with an unacceptable risk that have not yet occurred. For this, we generate the reachable component structures at runtime and analyze them with respect to risk. The system is modified such that component structures with an unacceptable risk are not reachable any more and are thus prevented.**

*Keywords*—**security and safety applications, self adaptive technologies, robust systems**

## I. Introduction

The value creation in today's technical systems is mostly driven by embedded software. Self-x techniques as an example of innovative functionality have become a major trend in engineering complex systems [1]. Self-x postulates that systems adapt autonomously to changes in the environment or the system itself, e.g., error occurrences. Self-x systems are often embedded real-time systems that interact with the real world, where they are employed in safety-critical contexts. Even in the case that the system does not contain any design errors, hazardous situations may be caused by random errors that happen, e.g., due to the wear of physical components. Consequently, these systems have to be analyzed with respect to potential hazards and risks.

Hazard analysis determines which combinations of random errors lead to hazards and the probability of the hazards' occurrences. The system developer uses this information to implement the system and in particular its software such that the risk, i.e., the probability of the occurrence combined with the severity of a hazard, is acceptable[1].

[1] Acceptability is the trade-off between the hazard's probability and consequences and the costs of reducing the probability of the hazard [2].

Self-x systems pose a challenge to safety analysis as they change their component structure during runtime. The structural change modifies the influence of hardware faults on the system and the occurrence probabilities of these faults. This affects the probabilities of hazardous situations. The severity that results from accidents may change as well, e.g., due to a changing amount of passengers in a car. Both parameters affect the associated risks. To guarantee safety, this analysis has to be applied to all component structures that are created in the self-x system. But not all component structures are known at design time, e.g., when the system connects to another system of which the system model is unknown at design time, e.g., the other system was made by an unknown manufacturer. Consequently, risk analysis has to be performed during runtime.

Cars that are driving autonomously, e.g., on a highway, are an example of Self-x systems. These cars can drive in a convoy in order to reduce air resistance and, thus, save energy. For this, they control their driving speed autonomously. Speed sensors measure the current speed of a car. Depending on this, the embedded software determines the required de-/acceleration to ensure the vehicle's drive speed. If at least one of the speed sensors fails, it propagates a wrong current speed to the embedded software. This results in a wrong de-/acceleration which, in turn, leads to the hazard *wrong speed*. A wrong speed may lead to the accident *collision*. The severity caused by this collision may be the injury of several people and severe property damage.

Standard development approaches for safety-critical computer systems require hazards to be identified and the associated severity to be defined in order to assess the systems risk [2]. Existing approaches of runtime analysis [3], [4], [5] focus on the detection of anomalies in the executed system behavior and try to lead the system back to its intended behavior. Detection cannot be applied if we want to prevent unwanted situations, e.g. unacceptable risks, before they actually happen.

In this paper, we present an extension of our runtime risk analysis [6]. That approach determines the risk of all future component structures at design time and prevents them from being created during runtime if their risk is unacceptable. That approach does not work for the aforementioned case when systems are connected during runtime which are not known at design time. In contrast to that previous work, we now compute hazard probabilities during runtime instead of using hazard probabilities which are pre-computed at design time.

Based on the system's current component structure, we compute each reachable component structure for a fixed number of subsequent structural changes. We then compute the hazard probabilities of the reachable component structures and combine them with the current severity encoded in numerical values to obtain the risks. If the risk of a reachable component structure exceeds the system's acceptable risk, the structural change that would result in this component structure is blocked.

The remainder of this paper is structured as follows. We first present the models that we use for modeling the component structure and structural changes in Sec. II. Our approach for risk analysis at design time follows in Sec. III. The risk analysis that is applied during runtime is presented in Sec. IV. Sec. V contains a discussion of related work. We conclude and give an outlook on future work in Sec. VI.

## II. System Models

In this section, we present the models that specify the systems structure and behavior. The models that we use in this paper are specified in MechatronicUML [7]. We describe the specification of the system architecture and the specification of the structural changes, called reconfiguration, in Sec. II-A and the behavior specification in Sec. II-B.

### A. *System Architecture*

In our approach, the system's software architecture is specified by an initial software component structure over a component model and a set of graph transformation rules for the specification of the architectural reconfiguration. MechatronicUML uses a component model where components communicate via ports and the component's behavior is specified by real-time statecharts -- an extension of UML State Machines which support more powerful concepts for the specification of real-time behavior. The components are connected via their ports by connectors. We call the resulting component structure a *configuration* [7].

We use UML deployment diagrams extended by hardware ports to specify and analyze the influence of hardware faults to the software components. Fig. 1 shows a deployment diagram of a simplified subsystem that controls the speed of an autonomous car. Software components are represented by rectangles and hardware components by boxes. Squares depict ports and arrows connectors. The orientation of the arrow specifies the direction of the communication. We will refer to this deployed component structure as *vehicle1*. This subsystem is a safety-critical part as a wrong speed can lead to harmful accidents like collision.

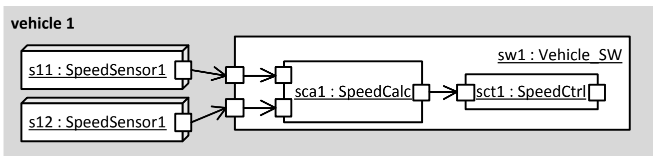

Figure 1. Deployment Diagram of the Speed Control Subsystem

The speed control subsystem consists of the component *sw1:Vehicle_SW* representing the vehicle's software and two speed sensors *s11:SpeedSensor* and *s12:SpeedSensor*. The speed sensors measure the vehicle's speed. The software compares and combines both speed values in *sca1:SpeedCalc* and subsequently sets the values for the engine controller in *sct1:SpeedCtrl*.

For the specification of reconfiguration, we consider configurations as graphs, i.e., components and ports as nodes and connectors as edges. We then model a reconfiguration as a *graph transformation rule* [8], [9]. A graph transformation rule consists of a left hand side and a right hand side. The left hand side identifies the part of the configuration in which the rule can be applied, i.e., the graph contains an isomorphic image of the left hand side as a sub graph. The right hand side defines the result of the application, i.e., the changes to be made to that sub graph such that it is an isomorphic image of the right hand side.

For a short hand notation, left and right hand side are depicted in one single graph using the notation «create» or «destroy» for components and connectors which are created or deleted during rule application.

Graph transformation rules change the system structure, i.e., create and delete components, ports, and connectors during runtime. They can be applied arbitrarily often. Additionally, new configurations are created by the connection of beforehand isolated systems as in our car convoy example. Thus, the concrete configurations that actually exist in the system are only known at runtime.

Fig. 2 shows the graph transformation rules that create a connection between two vehicles. Both graph transformation rules create a port that communicates to the vehicle in front and the vehicle in the back of the respective vehicle. During the creation of the ports, the connection between the vehicles is created as well.

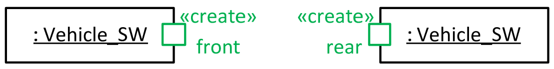

Figure 2. Graph Transformation Rules that Create a Connection between the two Vehicles

Fig. 3 shows a configuration which is the result of the graph transformation rules of Fig. 2 connecting two vehicles. It shows the connection between *vehicle1* and *vehicle2* when both vehicles drive in a convoy. Both vehicles run the same software, but they use speed sensors of different types – type *SpeedSensor1* in *vehicle1* and type *SpeedSensor2* in *vehicle2* – that have different probabilities of failing. Via this connection, *vehicle2* sends reference data, e.g., reference speed to *vehicle1*. *vehicle1* takes these data into account when calculating its target speed. Thus a failure in, e.g., *s21:SpeedSensor2* of *vehicle2* now also affects the hazard *wrong speed* of *vehicle1*.

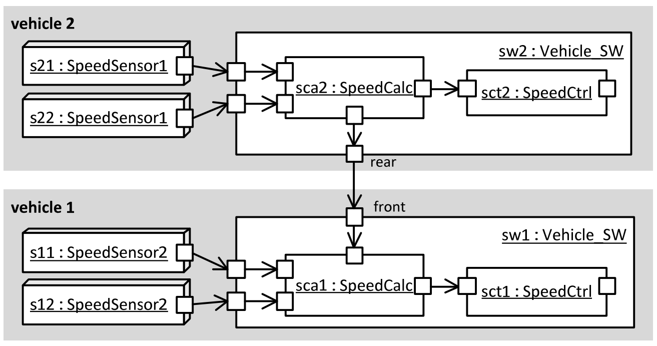

Figure 3. Component Instance Configuration in Convoy Mode

## B. System Behavior

We specify the real-time behavior of software components by *real-time statecharts* [7]. Real-time statecharts are an extension of UML State Machines which support more powerful concepts for the specification of real-time behavior. They contain clocks and constraints over these clocks. In this paper, we will use real-time statecharts without time. Reconfiguration is connected to the system behavior by attaching graph transformation rules to transitions of real-time statecharts as side effects.

Fig. 4 shows an excerpt of the real-time statechart that specifies the reconfiguration behavior of the component *sw1:Vehicle_SW*. The real-time statechart has four states. *noConvoy* is the state when the vehicle is not driving in a convoy. This is the initial state. In the states *convoyFront*, *convoyRear*, and *convoyInside*, the vehicle is driving as first vehicle, last vehicle, and inside a convoy. When switching between any of the four states, a port is either created or deleted by the graph transformation rule executed as a side effect of a transition. E.g., when switching from *noConvoy* to *convoyFront*, the side effect *{createRearPort()}* creates a port that communicates with the vehicle that is driving behind the current vehicle.

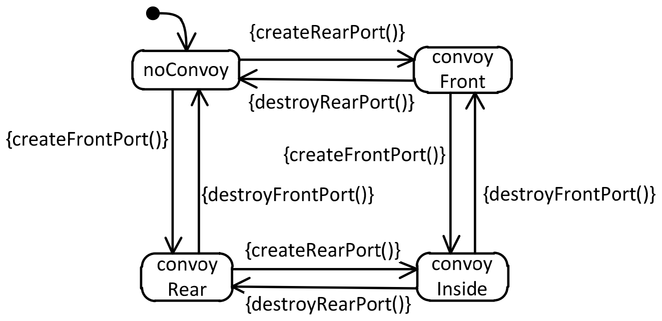

Figure 4. Real-time Statechart of the Vehicle Software

## III. Risk Analysis

The risk is the combination of the probability of a hazard and the severity associated to an accident that results from the hazard. Thus, for risk analysis we first compute the probability of a hazard by a hazard analysis. We then combine the probability with the severity value.

Our hazard analysis approach [10] computes hazards' probabilities based on configurations. In this section, we describe how the hazard analysis is performed for one configuration.

We follow the terminology of Laprie [11] by associating *failures* – the externally visible deviation from the correct behavior – to the ports where the components interact with their environment. *Errors* – the manifestation of a *fault* in the state of a component – are restricted to the internals of the component.

For each component, we determine the errors and the failures and build a failure propagation model manually as a set of fault trees formalized using Boolean logic, which relates failures at the ports of the components to internal errors. Internal errors are annotated with the probability of their occurrence. Failure propagation models for the connectors are automatically generated. Connecting the failure propagation models of all components by the connectors forms the failure propagation of a configuration. Using this failure propagation model, we compute cut-sets -- combinations of errors that make a hazard occur -- and probabilities of hazards.

Fig. 5 shows the failure propagation model of the component structure of Fig. 1. In each speed sensor, an error, e.g., wrong value can occur, illustrated by the errors $e_1$ and $e_2$. The failure $f_9$ at the outgoing port of the component *sct1:SpeedCtrl* occurs, if there occurs an error in either of the speed sensors. This is visualized by the symbol $\geq 1$ that represents a logical or.

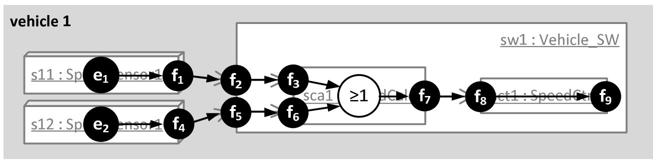

Figure 5. Failure Propagation Model of the Configuration of Fig. 1

We define a hazard in form of a Boolean formula that combines the outgoing failures of components. In our example, we define the hazard *wrong speed* by $wrong\_speed \Leftrightarrow f_9$. The speed sensors *s11:SpeedSensor* and *s12:SpeedSensor* each fail at a probability of $5 \cdot 10^{-5}$. The probability of the hazard *wrong speed* is (due to the logical or) thus approx. $10^{-4}$.

The risk of an accident is computed by multiplying the probability of the hazard with numerical values associated to the severity that results from the accident related to the hazard [2]. In our example, a collision that results from a wrong speed of *vehicle1* has a severity of 1 because *vehicle1* transports neither passengers nor cargo. The risk of *vehicle1* for the accident collision is thus $1 \cdot 10^{-4} = 10^{-4}$.

## IV. Safe Reconfiguration

We now show how we guarantee that configurations built during runtime do not have an unacceptable risk. For this, we extend the system architecture by components that execute the risk analysis at runtime and manage the component's reconfiguration, i.e., block reconfigurations in case the risk would become unacceptable. To make reconfigurations blockable, we extend real-time statecharts. These two model extensions are addressed in Sec. IV-A. On the other hand, we need to compute the possible configurations of the system during runtime, because we can only analyze dependencies between errors correctly if we know how components are connected. This is presented in Sec. IV-B.

## *A. Model Extensions*

Each autonomous component [2] is extended by the subcomponent *RiskManager*. Its subcomponent *Manager* manages the safe reconfiguration of its autonomous component with respect to risk, i.e., it blocks reconfigurations in case the risk of the subsequent configuration would not be acceptable. One autonomous component is chosen, of which the risk manager performs the risk analysis of the system and acts as the autonomous component's risk manager at the same time. In this case, the risk manager is extended by the subcomponent *RiskAnalyzer*. The risk analyzer can be deployed on any autonomous component. Leader election protocols [12] can be employed to determine at runtime which autonomous component will contain the *RiskAnalyzer* subcomponent. In systems where the computing hardware resources are limited, the risk analyzer may be deployed on an external unit, e.g., desktop computer.

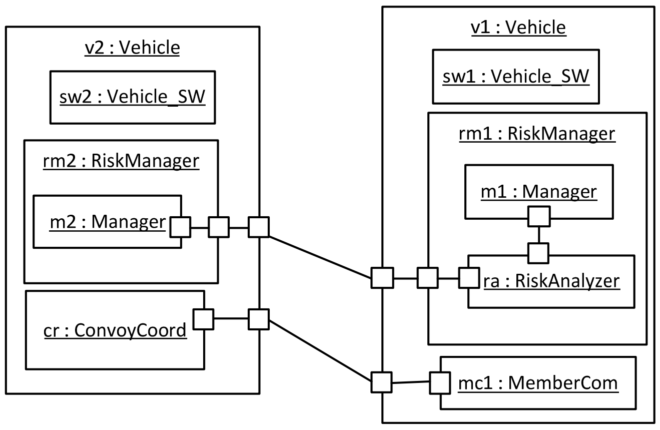

Figure 6. Component Structure Extended with Risk Coordinator Components

Fig. 6 shows a configuration of two vehicles driving in a convoy. *v2:Vehicle* coordinates the convoy, i.e., sets reference data, e.g., speed. *v1:Vehicle* drives as a member and performs the risk analysis.

For risk analysis, all autonomous components send their current configuration, graph transformation rules, and failure propagation models to the risk analyzer. The risk analyzer computes the reachable configurations of the system for a fixed number of steps from the current configuration and the graph transformation rules (cf. Sec. IV-B). For each reachable configuration, it computes the risks of all system hazards as described in Sec. III. Subsequently, each autonomous component receives the risk values and hazard probabilities associated with any of its reachable configurations. Based on this data, the risk manager of each autonomous component checks whether the risk of the subsequent configuration is acceptable before executing a reconfiguration.

[2] Autonomous components are components that are not embedded into higher level components. We call all other components ``subcomponent".

After each reconfiguration, the risk analyzer is updated about the changed configuration of the system by the risk manager of the autonomous component that has executed a reconfiguration. Every time the severity of a hazard changes, e.g., due to a changed amount of passengers, the risk managers update the risk values according to the hazard probabilities.

The risk analyzer not only needs to know which autonomous components are part of the system, but also how they are connected to each other. Each time an autonomous component wants to join the system, e.g., a vehicle intends to enter the convoy, the risk analyzer needs to be informed about which connections this component intends to establish to other autonomous components. With this ID the Risk Analyzer reconstructs which autonomous component is connected to which other autonomous components. The structure is needed for the hazard analysis (cf. Sec. III).

The new component requests an ID from the risk analyzer and from the components it wishes to connect to. This information is transmitted to the risk analyzer that judges whether the resulting risk of the system is acceptable or not. The communication between the new vehicle and the vehicle executing the risk analyzer is shown in Fig. 7.

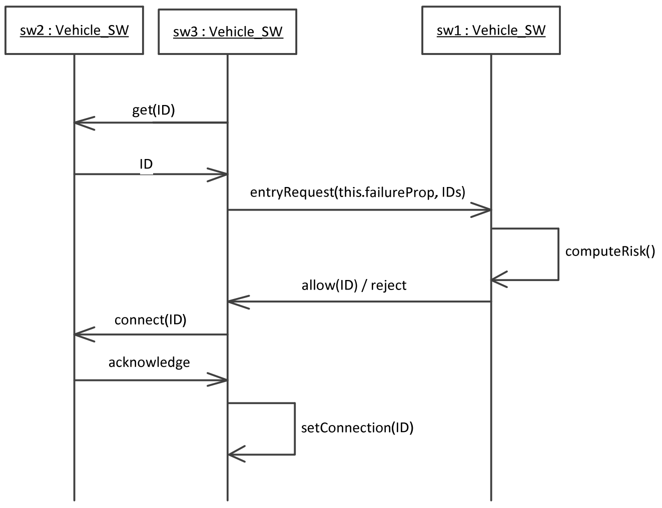

Figure 7. Communication for a Component Connecting to the System

In our example, the new vehicle, namely *vehicle3*, first requests the ID of the vehicle it wants to establish a connection to, namely *vehicle2*. After this, *vehicle3* sends its request to entry at *vehicle1* that acts as the Risk Analyzer. The Risk Analyzer either permits the connection or rejects the request. If the connection is allowed, both vehicles connect and *vehicle3* stores the ID of *vehicle2* for possible later reconfiguration requests.

In order to stop reconfigurations from being executed, we must block the transition that carries this reconfiguration as a side effect. For this, we introduced *blockable transitions* for real-time statecharts [6]. The engineer has to annotate whether a transition is blockable or not. These transitions are only executed, if the *RiskManager* has allowed this. Otherwise, they are blocked by the *RiskManager*.

In some cases, blocking transitions is not acceptable, e.g., reconfiguration in case of a component failure. Consequently, not all transitions that have a graph transformation rule attached can be blockable transitions. In order to still guarantee safe reconfigurations, we analyze all configurations that are reachable on paths between blockable transitions. If an unsafe configuration is reachable, the whole path is blocked by blocking its first transition which is a blockable transition. For further information we refer to [6].

## B. *Runtime Risk Computation*

The runtime risk computation consists of two steps. First, we compute all configurations of the system that are reachable from the current configuration by graph transformation rules. Second, for each reachable configuration we compute the probability of each hazard and combine it with the severity that is associated to this hazard. We take the highest risk value and compare it to the acceptable risk of the system in order to judge whether the configuration is safe or not.

For computing the reachable configurations, we perform a reachability analysis on the reconfiguration behavior. As described in Sec. II, the reconfiguration behavior consists of real-time statecharts that execute graph transformation rules as side effects of their transitions.

We compute the reachable configurations using the reachability analysis introduced in [13]. Based on the current system configuration and the set of graph transformation rules, the reachability analysis computes all possible successive configurations. The result is a labeled transition system whose states are configurations and whose transitions correspond to applications of graph transformation rules. The labels of the transitions are the names of the applied rules.

Fig. 8 shows the labeled transition system for the configuration of Fig. 1 and the graph transformation rules of Fig. 2. The labeled transition system consists of four configurations, namely *c1*, *c2*, *c3*, and *c4*. In the initial configuration *c1*, *sw1* and *sw2* do not have any ports. Then, either *sw1* may instantiate its rear port by executing the reconfiguration *createRearPort* or *sw2* may instantiate its front port by executing the reconfiguration *createFrontPort*. That leads to configurations *c2* and *c3*, respectively. After that, the second port may be created which establishes the connections between *sw1* and *sw2*. Of course, they are far more configurations reachable in the system indicated by the arrows leaving *c4*. Due to lack of space, we only present this excerpt.

For improving the efficiency of the analysis, we use a depth limited search and identify isomorphic configurations similar to GROOVE [14]. The depth limitation restricts the length of a path in the labeled transition system to a predefined number of states. Thus, we only investigate the next *n* configurations that are possible in the system.

A further reduction of the labeled transition system is achieved by identifying isomorphic configurations. If we reach a configuration which is identical up to isomorphism to a configuration that is already contained in the labeled transition system, we identify both configurations and do not investigate the isomorphic configuration a second time.

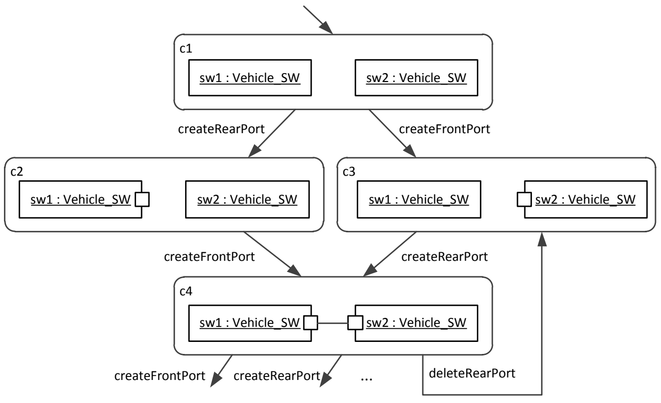

Figure 8. Labeled Transition System

In our example in Fig. 8, the configuration *c4* has been obtained by identifying isomorphic configurations. Applying reconfiguration *createFrontPort* to *c2* leads to exactly the same configuration which is obtained by applying *createRearPort* to *c3*. If we did not identify isomorphic configurations in our algorithm, the two configurations would be considered as two states which would lead to a larger labeled transition system.

For each reachable configuration, we compute the risk of each hazard as described in Sec. III. For our example configuration in Fig. 3 which is reachable from the initial configuration of *vehicle1* Fig. 1, we assume an acceptable risk of $5{\cdot}10^{-4}$. All speed sensors of *vehicle1* and *vehicle2* may emit failures. The difference between the speed sensors of both vehicles is their probability of failing. The speed sensors of *vehicle1* fail at a probability of $5{\cdot}10^{-5}$ and those of *vehicle2* at a probability of $5{\cdot}10^{-4}$.

When the vehicles are not connected, the probability of the hazard *wrong speed* is $10^{-4}$ for *vehicle1* (cf. Sec. III) and $10^{-3}$ for *vehicle2*. In our example, a collision of *vehicle1* has a severity of 1 and the risk of *vehicle1* for the accident *collision* is $10^{-4}$ (cf. Sec. III). The collision of *vehicle2* has a severity of 4 because it has loaded 4 passengers. In case of a collision there is the possibility of multiple injured persons or even fatalities. The risk of *vehicle2* for the accident collision is thus $4{\cdot}10^{-3}$.

Fig. 9 shows the failure propagation model of the two vehicles in convoy mode. The outgoing failure $f_9$ of *vehicle1* now also depends on the errors of the speed sensors of *vehicle2*, namely $e_3$ and $e_4$. This affects the probability of the hazard *wrong speed* of *vehicle1* which is now $(1-(1-10^{-4}){\cdot}(1-10^{-3})) = 11{\cdot}10^{-3}$. The risk of a collision of *vehicle1* is now $4{\cdot}11{\cdot}10^{-3} = 44{\cdot}10^{-3}$. We take 4 as the severity value as it is the highest severity value in the system. This risk is higher than the acceptable risk of this system as $44{\cdot}10^{-3} > 5{\cdot}10^{-4}$. Consequently, the reconfiguration connecting both vehicles must be blocked.

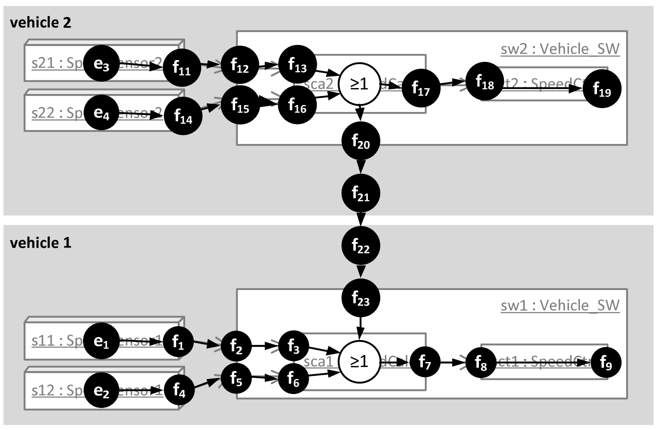

Figure 9. Failure Propagation Model in Convoy Mode

## V. RELATED WORK

Most approaches in the field of runtime safety analysis focus on the detection of anomalies in the executed behavior of the system and try to lead the system back to its intended behavior [3], [4]. We look into the reachable future states of the system and prevent it from unsafe states with respect to risk. However, this is only possible, because we know these potential anomalies beforehand and it only works for known hazards and risks.

Typically, for each safety property, that is to be tracked, a monitor is generated at runtime [3]. In our approach, we have one monitor that keeps track of the risk value of the system. The monitor is not generated but implemented offline. It becomes system specific by loading the system specific models at runtime.

The conditional safety certificates by Schneider et al. [15] model a relationship between required preconditions and safety guarantees in a fault tree like manner. During runtime, based on the current system properties, evidence is given whether the safety guarantees can be fulfilled. An adaptation is carried out accordingly. In contrast to our method, the analysis can only react to the current system properties. An analysis for several system states is not possible.

## VI. CONCLUSION

In this paper, we presented an approach for a runtime risk analysis for self-adaptive systems. In particular, we discussed the architectural extensions that are necessary to perform such runtime analysis as well as an algorithm for computing the risk. The algorithm applies a graph-based reachability analysis for computing all possible successive configurations of the system. The risks of these configurations are computed. The results of this analysis are then used to identify unsafe configurations which must be prevented.

In our future work, we will work on a concept for a compositional hazard analysis. Such an analysis eliminates the need for a central risk analyzer. Instead, each autonomous component computes its hazard probabilities by itself and the autonomous components only exchange hazard probabilities instead of behavioral models.

Further, performing a risk analysis during runtime opens up possibilities for risk management. We would like to reduce probabilities of hazard occurrences pro-actively by reconfiguring into a configuration with lower hazard probabilities.

## ACKNOWLEDGMENTS

This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

Christian Heinzemann is supported by the International Graduate School Dynamic Intelligent Systems.

## REFERENCES

[1] B. Cheng, R. Lemos, P. Inverardi und J. Magee, Software Engineering for Self-Adaptive Systems, Springer Berlin Heidelberg, 2009, pp. 1 - 26.

[2] N. Storey, Safety Critical Computer Systems, Addision Wesley, 1996.

[3] K. Sen, G. Rosu und G.Agha, „Online Efficient Predictive Safety Analysis of Multithreaded Programs,“ in *Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, Springer Berlin/Heidelberg, 2004, pp. 123-138.

[4] J. Rushby, Runtime Verification, Springer Berlin/Heidelberg, 2008.

[5] T. Vogel und H. Giese, „Adaptation and abstract runtime models,“ in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Capetown, South Africa, 2010.

[6] C. Priesterjahn und M. Tichy, „Modeling safe reconfiguration with the FUJABA Real-time Tool Suite,“ in *Proc. of the 7th International Fujaba Days*, Eindhoven, The Netherlands, 2009.

[7] S. Becker, S. Dziwok, T. Gewering, C. Heinzemann, U. Pohlmann, C.Priesterjahn, W. Schäfer, O. Sudmann und M. Tichy, „MechatronicUML - Syntax and Semantics,“ University of Paderborn, Paderborn, Germany, 2011.

[8] G. Rozenberg, Handbook of graph grammars and computing by graph transformation, River Edge, NJ, USA: World Scientifc Publishing Co., Inc., 1997.

[9] T. Eckardt, C. Heinzemann, S. Henkler, M. Hirsch, C. Priesterjahn und W. Schäfer, „Modeling and verifying dynamic communication structures based on graph transformation,“ in *Computer Science - Research and Development*, Springer, 2011.

[10] H. Giese und M. Tichy, „Component-based hazard analysis: Optimal designs, product lines, and online reconfiguration,“ in *Proc. of the 25th SafeComp*, Gdansk, Poland, 2006.

[11] A. Avizienis, J.-C. Laprie, B. Randell und C. Landwehr, „Basic concepts and taxonomy of dependable and secure computing,“ *IEEE Trans. Dependable Secur. Comput.,* pp. 11-33, 2004.

[12] A. Tanenbaum und M. v. Steen, Distributed Systems: Principles and Paradigms, Prentice Hall International, 2008.

[13] C. Heinzemann, J.Suck und T. Eckardt, „Reachability analysis on timed graph transformation systems,“ in *Proc. of the Fourth International Workshop on Graph-based Tools*, Enschede, The Netherlands, 2010.

[14] A. Rensik, „Isomorphism Checking in GROOVE,“ in *Graph-Based Tools (GraBaTs)*, Natal, Brazil, 2007.
57-60.

[15] D. Schneider und M. Trapp, „Conditional safety certificates in open systems,“ in *EDCC-CARS*, 2010, pp.