A New Modeling Approach for Automated Safety Analysis Based on Information Flows

Philipp Hönig and **Rüdiger Lunde**
Hochschule Ulm
University of Applied Sciences
Institute of Computer Science
e-mail: {hoenig, r.lunde}@hs-ulm.de

Abstract

A central challenge in automating failure-related analysis tasks during a product life-cycle (such as safety analysis and diagnosis) is the choice of an appropriate level of abstraction under which the system is considered. In this paper failure effects within a practical system including a control loop are studied. Existing approaches using quite different levels of abstraction are compared and partly evaluated. The results motivate a new modeling approach which we call *smartIflow*. We consider components as finite state machines and provide new concepts for undirected information exchange between the components and build-in capabilities for flow direction determination. Interesting parts of a model for the example system demonstrate the syntax. Based on these example code fragments important concepts of the new language are discussed in detail. Though no implementation yet exists, it is shown, that this approach has the potential to overcome several limitations of existing approaches.

1 Introduction

Safety-critical systems are systems where failures can lead to substantial material and environmental damage or even to the loss of life. Such systems can be found in various domains such as control systems in aircraft, medical devices, or assistance systems in cars. It is essential that the functionality of such systems can be guaranteed even in critical situations. Therefore various analysis techniques are applied during the development process of those systems to ensure system safety. Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) are probably the best-known analysis techniques. The problem with traditional analytical techniques is that most of the tasks must be done manually by engineers. Those traditional techniques consume a lot of time and effort, which results in higher development costs. The increasing complexity of systems, shorter product cycles and cost saving in almost all areas require new concepts and software solutions for safety analysis in an automated way.

Model-based safety analysis (MBSA) as described in [1] seeks to overcome these problems by using a formalized and computer-understandable system model for the various analysis tasks and also for the development. The incorporated knowledge of models can be very diverse. Models can for example include structural information, information about failure behavior or functional behavior. The so called extended system models, which include both, nominal and failure behavior, can be used for automating parts of the safety assessment. The knowledge representation of the model can also vary widely:

- Highly abstracted system models can be created early in the development process where the exact properties of the components are not known. These models can be used to get very early rough information about the system safety, but they are not detailed enough to get specific signal values or information about the timing.
- Physical models contain more precise information about the components used in the system. With these models, detailed information about timing can be gathered, or the signal value at a specific port can be analyzed. Getting widespread information about safety leads to high computational effort.

In the last decades several representation formalisms have been developed, and some of them even are integrated in commercial tools (Autosteve [2], HiP-HOPS [3], Rodon [4] or AltaRica [5] to name a few). A central challenge in the automation of safety and reliability analysis is the choice of an appropriate abstraction level. Either the abstraction level is too high and some effects can't be discovered or the models are too detailed which results in high modeling effort and subsequently to high computational effort. This is also the main problem behind the existing approaches.

In this paper we propose a new modeling formalism based on information flows which we call *smartIflow*[1]. We try to find an abstract perspective which supports reasoning about faults and their effects through the whole product life cycle, starting early at design time and finally including diagnosis. Since safety assessment is performed early in the life cycle of a technical system, we first want to demonstrate the applicability of our models to safety analysis. In this paper we investigate how our modeling formalism can be used for automated safety analysis. However, our long-term goal is to use these models for diagnosis, too.

This paper is organized as follows. In Section 2 some existing approaches and their principal limitations will be described briefly. In Section 3 an example system is introduced, and ideas for new modeling approaches are developed based on a discussion about selected relevant failure situations. In Section 4 our proposed modeling formalism

[1]State Machines for Automation of Reliability-related Tasks using Information FLOWs

will be presented in detail, and afterwards it will be described how the situations in the example system are handled by it. A suitable simulation algorithm will be sketched in Section 5. Finally a short conclusion will be given.

2 Related Work

Current approaches differ in many aspects. One aspect is the *level of abstraction* on which the behavior is specified. Quantitative models will obviously deliver more detailed results than qualitative models. However, at the same time the modeling effort and also the computational effort will be much higher. And not all failure situations are predictable on the very deepest level of abstraction. Another distinguishing criterion for component-oriented modeling approaches is the type of *connection modeling*. The connections between the components can be directed as in Simulink[2] or undirected as in Simscape[3]. Finally, there are different ways how *failure behavior* can be modeled. Modeling formalisms can roughly be divided into the categories Failure Logic Modeling (FLM) and Failure Injection (FI) [6]. In Failure Logic Modeling only the failures and their propagation through the components of the system are modeled on a high level of abstraction. In FI, a model describing the nominal behavior of a system is enriched with information about component failures. Behavioral descriptions in FI can, but not necessary need to be, based on physical laws using a rather low level of abstraction.

HiP-HOPS [3] (Hierarchically Performed Hazard and Operability Studies) developed by Papadopoulos et al. is a FLM-based modeling formalism using an extremely high level of abstraction. In HiP-HOPS for each component a fault model is created that specifies how a component responds to internal failures or failures generated by other components. The connections between the components are directed and used as channels for the failure propagation. A component in HiP-HOPS is characterized as a set of expressions which describes the causes of output failures in terms of logical combinations between internal failures and failures on the input of a component. There are some predefined failure classes which can be used in these expressions, like omission, commission, or timing.

Failure propagation is limited to one failure per connection which means that only the deviation of one physical quantity (e.g. flow) can be taken into account. The components in HiP-HOPS contain no state variables. Therefore, it is not possible to specify the failure propagation depending on the operation mode. Since the connections between the components are directed, failure situations in which the causal flows reverse, sometimes even completely unexpectedly, can't be handled.

In addition to HiP-HOPS, there are a number of other interesting approaches and concepts. Failure Propagation and Transformation Notation (FPTN) [7] is very similar to HiP-HOPS, however instead of a textual representation, FPTN uses a graphical notation to specify the failure behavior of a component on a high level of abstraction. AutoSteve [2] is a commercial tool that performs automated design analysis of electronic circuits based on functional abstraction and qualitative simulation. Rodon [4] is another commercial tool that uses a component-oriented and object-oriented language to model the behavior of the system. Behavior prediction is based on constraint propagation, and a Reason Maintenance System (RMS) enables focused search for possible causes of undesired events. Struss and Dobi presented in [8] an approach to automated safety analysis based on qualitative models. They use deviation models [9] to describe incorrect behavior. A deviation model captures the deviation from the reference state on a high level of abstraction by using only a few signs. The AltaRica [5] formalism describes systems with a set of nodes. Each node has a finite number of state and flow variables. State transitions are triggered by events.

In addition, there are a number of approaches which are fundamentally based on model checking. In the approach of Joshi et al. [1], the nominal system model is enriched with failure behavior. This extended system model is translated into the input language of the SMV model checker. The safety requirements are specified in some temporal logic (CTL/LTL), and the model checker verifies whether the model fulfills these requirements. If the model doesn't fulfill the requirements, the model checker automatically generates a counterexample. These approaches have been investigated in [10]. The following table summarizes the discussed approaches:

Approach	**Level of Abstraction**	**Connection Modeling**	**Failure Behavior**
HiP-HOPS	extremely high	directed	FLM
FPTN	very high	directed	FLM
Joshi et al.	low-high	directed	FI
AutoSteve	high	undirected	FI
Rodon	low	undirected	FI
Dev. Models	high	undirected	FLM
AltaRica	low-high	undirected	FI

Using physical system models for safety analysis delivers very detailed results. However, the search space can grow very fast, and thus we will often need more abstraction to reach a satisfactory level of completeness with acceptable computational efforts. HiP-HOPS is the extreme opposite of physical modeling. The modeling formalism is very simple but at the same time also very limited. We are searching for a modeling formalism whose level of abstraction lies between HiP-HOPS and physical modeling. Anyway, without practical studies, it is not clear which of the listed principal limitations turn out to be grave in practice.

3 Discussion Based on a Practical Example

At first glance, HiP-HOPS seems to be a very attractive high abstraction level approach to automated safety analysis and related applications. Naturally, the simplified view on systems under analysis comes with limitations regarding the accuracy of predicted cause effects. But how are modeling costs and accuracy of results balanced in practice?

We will now take a look at an example system containing a control loop. By focussing on selected relevant faults and their effects we try to get a practical understanding of the limitations of HiP-HOPS and other approaches. From this analysis, suitable concepts will be derived to overcome those limitations.

Figure 1 shows the structure of a simple cooling system. The cooling system is responsible for cooling down

[2]`www.mathworks.com/products/simulink`
[3]`www.mathworks.com/products/simscape`

hot HNO_3 (nitric acid) to a safe temperature T. This is achieved by using a heat exchanger that takes cool water to cool down the hot fluid. For energy saving reasons, the cooling water supply can be reduced. A controller measures the temperature of the out-flowing fluid and controls valve V2 depending on the measured data. The aim of the controller is to keep the temperature of the nitric acid in a predefined range. A safety system consisting of a flowmeter and valve V1 blocks the hot fluid if the amount of inflowing water drops under a certain threshold. The cooling system operates in one of the following operation modes:

a) Cooling mode: Normal operation mode.
b) Maintenance mode: Valve V1 is closed and Circulation Pump is switched off.

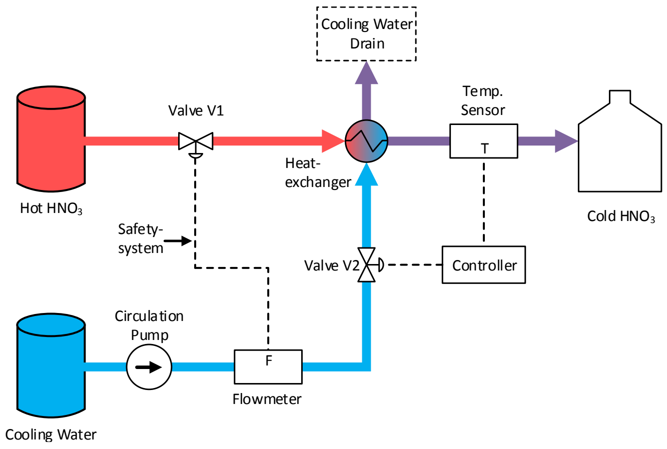

Figure 1: The cooling system (Source: [11])

The cooling system contains some non-trivial failure situations which a modeling formalism should be able to explain. Due to the lack of space we only take a detailed look at the three most interesting ones:

1. A StuckAtClosed-failure of valve V2 will lead to the situation that no or not enough cooling water will flow. The flowmeter as part of the safety system detects this situation and closes valve V1 to avoid the rising of the fluid temperature above a critical threshold. In this case, the failure of V2 has an indirect influence on the flowmeter and thus also on V1.
2. An internal failure inside the controller leads to an undetermined flow of the cooling water, because the position of the valve is not set properly. Therefore there might not be enough cooling water to cool down the hot fluid.
3. The circulation pump is provided with too much voltage which results in a high pressure of the cooling water. The heat exchanger is unable to deal with the pressure which results in a leakage.

To enable automated safety analysis, the system behavior needs to be modeled in a formal and computer-understandable language. In general, systems are composed of multiple interconnected components. To make behavioral models reusable *component-oriented modeling* is a key. Component type models specify the common behavior of components which are equal with respect to the chosen abstraction level. Usually, those type models provide typed ports which define the interface for interaction with other components. System models (which themselves can be used as component type models, on a higher level within a component hierarchy) are created by connecting ports of instantiated type models. For component-oriented modeling, it is essential to formulate behavior in a context-independent way. Obviously, with increasing level of abstraction, this becomes more and more difficult. HiP-HOPS for example annotates Simulink components with failure propagation information. Hence, behavior is specified on component level. But since propagation rules for heavily abstracted failures are by their nature context-dependent, different specifications are needed for the same component type. HiP-HOPS solves the problem by assigning the annotations to component instances instead of component types.

The StuckAtClosed-failure of valve V2 in the first use case has extensive consequences to all components that are attached along this line. On one hand the heat exchanger receives not enough cooling water, but on the other hand the flowmeter detects that there is no flow of cooling water and therefore valve V1 is closed. Using directed connections for exchanging information between the components like in HiP-HOPS leads to the problem that every time the cause-effect relationship may reverse, artificial connections must be introduced. Such situations often occur unexpectedly. Therefore it is almost impossible to capture all these situations in the model. Actually, such situations should be detected automatically and not manually by the safety engineer. In our example, it is necessary to create an additional connection from valve V2 to the flowmeter. These additional connections lead to very complex models which are hard to maintain. To avoid this additional modeling effort, the components must be connected using *undirected* connections, and the flow must be determined dynamically based on the network structure and component behavior. Since quantitative flow predictions are not in our focus here, it might even be sufficient to integrate a simple flow analysis mechanism which combines state-dependent connection information with knowledge about a limited set of built-in component behaviors (e.g. drain, source, bipolar source).

In the early stages of the development process, only a sketch of the system architecture is available. The detailed behavior and the properties of the individual components are unknown. Therefore, it is not possible to model the system on a quantitative level and to predict the exact behavior. In addition, calculating the system behavior on a quantitative level would also require to test all possibilities, which quickly leads to state space explosion. Thus the behavior of the components need to be modeled *qualitatively* on a high level of abstraction. HiP-HOPS uses connections as channels to propagate failure effects like "too much of this" (commission) or "not enough of that" (omission). But propagation is limited to just one effect per connection. In the example system, this is insufficient, since deviations of several physical quantities, namely temperature and flow, play an important role in the failure model. In addition, the available failure classes are sometimes insufficient. For example, it is very hard to explain an excessive temperature in terms of omission respectively commission.

Another possibility to abstract from the physical conductor is to consider the connections between components as information channels. The information shared by the components can be very diverse. On the one hand it can be in-

formation about the flow (low/high) and on the other hand also about the temperature (cold/hot). Each component is able to read and to modify the information on the channel the component is connected to. Depending on the shared information, a component may change its operational mode or modify (add, remove, update) the information on the flow.
In the second use case the internal failure of the controller leads to an uncontrolled flow of the cooling water, and thus the temperature of the fluid will be undetermined. In this case it is not enough just to talk in terms of "high flow" or "no flow", because the controller (and the valve) can behave in a multitude of faulty ways. For example, the controller might cause the valve to open (close) completely or to be stuck in an intermediate position. To be correct, it would be necessary to consider all failure modes in the component description of the controller, which results in a high modeling effort. A similar problem occurs when modeling the nominal behavior of the controller. It is very difficult to reproduce the behavior correctly on an abstract level, because it can lead rapidly to the situation that the actual behavior is replicated one-to-one in the model. In many cases it is not necessary to reproduce the behavior in any way, but it is more important to know whether the output is correct. Instead of trying to reproduce the behavior of the controller, it could be sufficient to indicate that the controller has a malfunction and thus the state of the valve is not regulated. The information about the correctness of an output must be propagated through the information channels, as well. Though we can not make quantitative statements about the temperature of the liquid, we at least know that it is not regulated. Deviation models seem appropriate for this purpose.

In the third use case, the circulation pump is supplied with too much electricity, which results in overpressure in the conduit to the heat exchanger. Since the coupling of the components is not designed for such high pressure, this may inflict a leakage somewhere. We can not predict where exactly the leakage is occurring, because to do so the exact characteristics of the components must be known, which is not the case in the beginning of the development. Exact results can be obtained only by testing with real hardware. In the early stages of development, it is entirely sufficient to know which components might be affected and which effects arise in the respective situations. Non-deterministic behavior can't be handled in HiP-HOPS. One possible way to get a grip on the non-deterministic behavior is to inform all affected components by sending a property downstream across the flow, starting at the pump. Components downstream listen to the property and choose between two possible reactions: Stay in OK mode or change to leakage failure mode.

Usually a component has more than one failure mode. For example, the valve in the example system might be stuck in open, closed or in any other intermediate position. Obviously the behavior in the different failure modes varies widely. Therefore, the modeling formalism should provide a mechanism to distinguish between failure modes of a component and the according behavior. This concerns not only the failure behavior of a component, but also the nominal behavior. In HiP-HOPS the components have no states which becomes a problem when the system operates in the maintenance mode. If the system is operated in the maintenance mode, the circulation pump is turned off, and hence there is no flow of cooling water. The inflow of the hot liquid is also interrupted. In this case, a failure situation in valve V2 is non-critical due to the fact that there is no flow of hot fluid. HiP-HOPS will indicate this situation as a hazard because there is no distinction between the two operation modes. Therefore, the behavior of a component must be described depending on the internal state. The state of a component can be changed through internal or external events.

4 Automated Safety Analysis Based on Information Flows

Based on the results of the analysis of the practical example we developed a new modeling formalism called *smartIflow*. In this section we want to present our conceptual description language and the concepts behind it. The component descriptions of the controller and valve from the practical example might look as follows:

```
class Valve {
   Ports:
      Fluidal p1, p2;
      LogicalInput in1;
   States:
      Enum[Open, Closed, Controlled,
         Uncontrolled] s = Controlled;
      Enum[Ok, StuckAtOpen,
          StuckAtClosed] fm = Ok;
   Transitions:
      when(in1.val == Open && in1.dev == False &&
           fm == Ok || fm == StuckAtOpen) {
         s = Open;
      }
      ...
      when(in1.val == Controlled &&
           in1.dev == False && fm == Ok) {
         s = Controlled;
      }
      when(in1.dev == True) {
         s = Uncontrolled;
      }
   Behavior:
      if(s == Open) {
         connect(p1, p2);
      }
      if(s == Uncontrolled) {
         connect(p1, p2,
            [flow.val=Controlled, flow.dev=True]);
      }
      if(s == Controlled) {
         connect(p1, p2, [flow.val=Controlled]);
      }
}
class Controller {
   Ports:
      LogicalInput sensorValue;
      LogicalOutput outputValue;
   States:
      Enum[Ok, Failure] fm = Ok;
      Enum[Controlling, UndetectedInputFailure,
            FailSafe] om = Controlling;
   Transitions:
      when(sensorValue.dev == False) {
         om = Controlling;
      }
      when(sensorValue.dev == True) {
         om = UndetectedInputFailure
            or FailSafe;
      }
```

```
    Behavior:
        set(outputValue, [val=Controlled]);
        if(fm == Failure ||
           om == UndetectedInputFailure) {
           set(outputValue, [dev=True]);
        }
}
```

The basic idea behind our approach is to consider each component in the system as a finite state machine. We provide mechanisms for undirected information exchange between the components and for flow direction analysis. In fact, our approach follows the principle of Discrete Event Systems (DES). A DES is characterized by a discrete state space that changes only at a discrete set of points in time [12]. While traditional DES-based approaches such as the methodology of Sampath et al. [13] use manually defined events to describe the interaction between the components, in our approach the components exchange signal values instead of events. Events are generated within the receiving component based on signal changes. Thus, one essential distinguishing criterion is the way in which the components interact. In addition our modeling formalism is completely component-oriented.

Keep in mind that this is only a first draft of our modeling formalism; there are still some open questions. Nevertheless we want to take a closer look on the concepts used in our modeling formalism.

4.1 Compositional Modeling

The proposed language is component-oriented and object-oriented (though inheritance is not discussed here). For each component type a separate class is created. Similar components only need to be modeled once, because the classes can be instantiated several times. Component instances can be part of other components as well, to build hierarchical systems. A class consists of a unique name and a set of *sections*, where each section describes a different aspect of the component. The following sections can be used in a component description class: *States*, *Ports*, *Components*, *Transitions*, and *Behavior*.

4.2 Ports and Connections

The component-oriented perspective considers systems as a set of components that are interconnected. Connection points between the components are called ports. A port declaration consists of a type and a name which must be unique within a component. There will be a set of predefined port types for the most common domains (e.g. electrical, fluidal or mechanical), but it is also possible to define custom port types. Directed connections can also be created by using special port types, namely *LogicalInput* and *LogicalOutput*. This kind of ports is useful if the flow is restricted to one direction (e.g. see valve model). Special built-in elements for sinks and sources are available to determine the flows in the system. Connections between the components are modeled as passive channels, which means that information exchanged by the components is not modified during the transmission [14].

In our example, the component description of the valve contains three ports: Two ports of type *Fluidal* where the liquid is flowing through, and one port of type *LogicalInput* where the control signal is received.

4.3 States

As already said, the behavior of a component depends on the operational mode. In order to capture the states of a component, each component can include one or more state variables. States variables are private, which means that they are only visible inside the component itself. The definition of a state variable consists of a finite domain (typically an enumeration of symbolic values), a name, and an initial value. For instance, the component characterization of the valve contains two state variables: one for nominal and one for the failure behavior. The valve can be in state *Closed*, *Open*, *Controlled*, or *Uncontrolled*. The behavior of the component depends on its current state.

4.4 Transitions

Components change their internal states based on the information shared with other components. These state transitions could either be integrated into the behavior description or be modeled separately. Although integration into the behavior description would lead to a more compact representation, we decided to separate the state transitions from the behavior, for the sake of clarity. In the section *Transitions* it is described how a component changes its state. State transitions are specified by when-clauses followed by assignments to at least one state variable. A when-clause contains a logical expression which consists of comparisons between symbolic values and all kinds of variables including propagated properties. The syntax for relational operators (==, !=) and logical operators (||, &&, !) is well known from programming languages like Java. During simulation transitions are triggered when the value of the expression changes from false to true. The *Transitions* section encapsulates all aspects of indeterminism. Different semantics (synchronous/asynchronous transition models) can be used. Additionally, state variable assignments can be indeterministic too (see the second transition in the controller example).

4.5 Qualitative Behavior Description

Our analysis of the practical example has shown, that the level of abstraction of HiP-HOPS is too high and inflexible. In *smartIflow* we describe the behavior on a lower level of abstraction by focussing on the flows between the components and the information they exchange with each other. As already described, the behavior of a component depends on its internal state. Therefore, the behavioral description of a component consists of a set of conditional branches, in which the behavior of the respective state is described. A condition is composed of logical combinations between state variables and their characteristics. Here, the same relational operators and logical operators can be used as in the state transitions. In the conditional statements of the behavior description only state variables are permitted. A conditional branch that is always true (`if(true)`) can be omitted. The functions inside such a block can be declared directly in the section *Behavior*.

The behavior of a component is described in terms of modifications to the network structure and property publication through the network. Two ports can be connected by the function *connect*. This allows a flow through the component. If there should be no flow, the connect statement can be omitted. The behavior of a valve can be easily modeled by connecting the two ports in the state *Open*, while in the state *Closed* the ports are disconnected which interrupts the flow. The flows in the system will then change according to the component's state.

Only specifying the flow is sometimes not sufficient. In our example, the valve is controlled by the controller. We do not know exactly how the valve will behave, but we might know whether the input from the controller is correct or not. For this reason, it is possible to enrich the flow with additional properties, which can be read by subsequent components within their transitions sections. A property specification has the structure "*path=value*". The path specifies the quantity for which the information is specified (e.g. the pressure at a port). A path consists of a set of path elements separated by "." and a closing element which can be either *val*, *dev*, or *dir*. The closing element *val* is used to specify physical quantities or logic signals. *dev* is used to specify a deviation from the nominal behavior. Property name *flow.dir* is reserved for the built-in flow direction determination. Paths can also be empty, which means that they only consist of a *val* or *dev* element. To enrich the flow with additional information, one or more properties can be provided to the *connect* function. Additionally, function *set* allows to publish information without network topology modification.
In the example system the valve will change its state to *Uncontrolled* if the output of the controller deviates from the nominal behavior. Therefore, the flow through the valve will also deviate, which is indicated by the property *flow.dev=true*. For the state *Closed* there is no behavior specified since a closed valve blocks the flow on both ports.

5 Analysis Algorithm

We plan to test different analysis algorithms. A possible starting point could be a state transition tree exploration algorithm as sketched below:

```
 1: open-nodes = empty set
 2: closed-states = empty set
 3: open-nodes ← node for initial system state [4]
 4: while open-nodes is not empty do
 5:    node = remove a node from open-nodes
 6:    if state(node) not in closed-states then
 7:       reconfigure network according to state(node)
 8:       determine flows and propagate properties
 9:       compute possible next states
10:       for each possible next state do
11:          open-nodes ← new corresp. successor-node
12:       end for
13:       closed-states ← state(node)
14:    end if
15: end while
```

Since we are interested in all possible system reactions to certain failures, the outcome of behavior prediction cannot just be a sequence of (possibly time-stamped) system states, as it is the case in typical simulation tools. Instead, a tree of nodes is created in which each node corresponds to one system state, and each successor-link to one possible state transition (see Figure 2). The root of the tree corresponds to the initial system state, and leafs to situations, whose succeeding situations either do not exist (terminal states) or can already be looked up somewhere else in the tree.

[4]In this context "←" stands for an add operation

The presented algorithm is in principle depth-first search (DFS). The variable *open-nodes* marks parts of the tree whose successors still need to be analyzed. The variable *closed-states* is used to avoid loops and thus to guarantee termination. Behavior prediction basically consists of three main steps, namely network reconfiguration, flow calculation, and successor state computation.

In the network reconfiguration step the commands in the *Behavior* sections are executed depending on the actual values of the state variables. Commands can establish additional connections and assign properties to connections and ports.

In the flow calculation step, the directions of flows are determined using the current connection structure and knowledge about build-in primitives. It is assumed that a flow will always select the shortest path from a source to a sink. Properties are then propagated through the network, partly based on the analyzed flow directions. We take into account that this simplified view can lead to ambiguities.

In the last successor state computation step, all possible state transitions are evaluated. The set of all possible successor states depends on properties shared with other components, the direction of the flows through the component ports, and the actual values of component state variables. Since we consider the system behavior on a high level of abstraction, it is of course impossible to get detailed information about the timing.

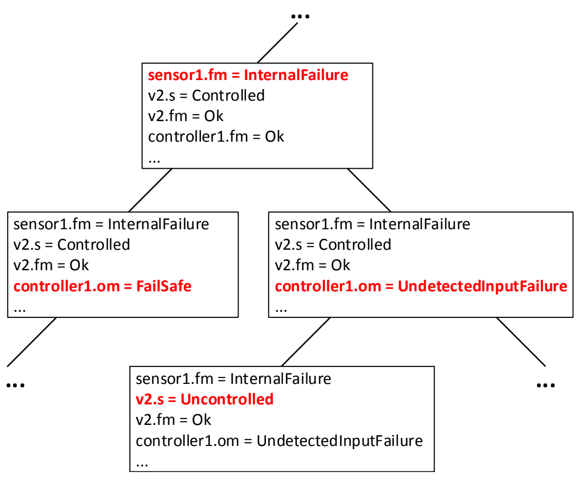
...
sensor1.fm = InternalFailure
v2.s = Controlled
v2.fm = Ok
controller1.fm = Ok
...
sensor1.fm = InternalFailure
v2.s = Controlled
v2.fm = Ok
controller1.om = FailSafe
...
sensor1.fm = InternalFailure
v2.s = Controlled
v2.fm = Ok
controller1.om = UndetectedInputFailure
...
sensor1.fm = InternalFailure
v2.s = Uncontrolled
v2.fm = Ok
controller1.om = UndetectedInputFailure
...
...
...

Figure 2: Excerpt of the state tree of the cooling system

6 Conclusion

Our initial goal was to develop a formalism for describing failure behavior of systems on an as-high-as-possible level of abstraction without loosing the advantages of context-free component-oriented modeling, and without loosing too much prediction power. We started with HiP-HOPS, a modeling formalism that focuses on failure propagation using an extremely high level of abstraction. The experiments with the practical example showed that approaches relying solely on failure propagation lead to hardly maintainable models with limited options for reuse. In fact, HiP-HOPS binds failure propagation models to instances of components instead of component types. Therefore, we identified several concepts to specify situations which HiP-HOPS is unable to describe. We also proposed a new formal language called *smartIflow* that integrates these concepts. Although the resulting component descriptions grew larger than intended, still the modeling effort is on an acceptable level.

At first glance it seems that we ended up with our formalism almost at the abstraction level of AltaRica, but there are some essential differences. In AltaRica, information exchange between components is based on synchronized typed variables which are called flow variables. Our approach also relies on typed variables (which we call ports), but the type (e.g. *Fluidal*) does not limit the propagated values. This leads to low coupling between senders and receivers within a network. Property propagation and flow direction determination together enable our formalism to send informations downstream in the direction of the actual flow. We believe this feature to be crucial for handling situations with unforeseen flow direction changes.

7 Future Work

Up to now, we focused on theoretical experiments with our modeling formalism. To verify the practical applicability, one of the next steps will be the implementation of a prototypical simulation framework. We also plan to extend our modeling language by additional aspects of object oriented languages (e.g. inheritance, visibility modifiers or generic classes). Finally, more challenging applications will be used to validate our formalism on a larger scale.

References

[1] Anjali Joshi, Mike Whalen, and Mats P.E. Heimdahl. Modelbased safety analysis: Final report. Technical report, 2005.

[2] Chris Price. Autosteve: Automated Electrical Design Analysis. In Werner Horn, editor, *ECAI*, pages 721–725. IOS Press, 2000.

[3] Yiannis Papadopoulos and John A. McDermid. Hierarchically Performed Hazard Origin and Propagation Studies. In *Proceedings of the 18th International Conference on Computer Computer Safety, Reliability and Security*, SAFECOMP '99, pages 139–152, London, UK, UK, 1999. Springer-Verlag.

[4] Karin Lunde, Rüdiger Lunde, and Burkhard Münker. Model-Based Failure Analysis with RODON. In *Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy*, pages 647–651, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

[5] André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The AltaRica Formalism for Describing Concurrent Systems. *Fundam. Inf.*, 40(2,3):109–124, August 1999.

[6] O Lisagor, J A McDermid, and D J Pumfrey. Towards a Practical Process for Automated Safety Analysis. 2006.

[7] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey. Towards Integrated Safety Analysis and Design. *SIGAPP Appl. Comput. Rev.*, 2(1):21–32, March 1994.

[8] Peter Struss and Sonila Dobi. Automated Functional Safety Analysis of Vehicles Based on Qualitative Behavior Models and Spatial Representations. In *The 24th International Workshop on Principles of Diagnosis (DX-2013).*, pages 85–91, 2013.

[9] Peter Struss. Deviation Models Revisited. In *In: Working Papers of the 15th International Workshop on Principles of Diagnosis (DX-04).*, 2004.

[10] Philipp Hönig. Automated Safety Analysis of Technical Systems based on Component-Oriented Models. Master's thesis, Hochschule Ulm, University of applied Sciences, Ulm, 2013.

[11] Eike Schwindt. Gefahrenanalyse mittels Fehlerbaumanalyse, 2004.

[12] Jerry Banks, John S. Carson II, Barry L. Nelson, and David M. Nicol. Discrete-Event System Simulation. PEARSON, 2010.

[13] M. Sampath, Raja Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *Control Systems Technology, IEEE Transactions on*, 4(2):105–124, Mar 1996.

[14] Peter Struss. Modellbasierte Systeme und qualitative Modellierung. In Günther Görz, Claus-Rainer Rollinger, and Josef Schneeberger, editors, *Handbuch der Künstlichen Intelligenz*. Oldenbourg, 2000.