Modeling Technical Systems with smartIflow for Safety Related Tasks

Philipp Hönig
Hochschule Ulm
University of Applied Sciences
Institute of Computer Science
hoenig@hs-ulm.de

Rüdiger Lunde
Hochschule Ulm
University of Applied Sciences
Institute of Computer Science
r.lunde@hs-ulm.de

Florian Holzapfel
TU München
Institute of Flight System Dynamics
florian.holzapfel@tum.de

ABSTRACT

SmartIflow (State Machines for Automation of Reliability-related Tasks using Information FLOWs) is a modeling formalism for automating safety related tasks. It considers systems on a quite high level of abstraction without losing too much predictive power. Since existing approaches often lack in integration into the development process, we present a new method that allows safety engineers to model in their familiar environment. Simulink, Simscape, and Stateflow offer an optimal platform to visualize and edit smartIflow models. This paper describes the graphical notation of smartIflow components in Simulink and the translation into the original language. The practicability of our new graphical modeling approach is shown by an example system that we have modeled successfully.

Categories and Subject Descriptors

B.8.1 [**Hardware**]: Reliability, Testing, and Fault-Tolerance; I.6.2 [**Computing Methodologies**]: Simulation Languages

General Terms

Algorithms, Reliability, Languages

Keywords

Model-Based Safety Analysis, smartIflow, AltaRica, HiP-HOPS, Simulink, Simscape, Stateflow

1. INTRODUCTION

Assistance systems in cars are primarily designed to improve safety and comfort. However, a failure in such systems can lead to critical situations like abrupt unintentional steering movements or brake actions. Such systems are called safety-critical [4], since an error can lead to high material damage or even to loss of life. Therefore, it is an essential task to ensure system safety even in critical situations. This is achieved by applying several analysis tasks during the development process. Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) are probably the most famous methods for safety assessment. The problem with these traditional analysis techniques is the only partly available automation, which makes the analysis time consuming and also error-prone (especially for more complex systems). Model-based safety analysis (MBSA) as defined in [3] tries to overcome this problem by using a formalized and computer understandable model of the system to automate the analysis process. There are already a lot of approaches to MBSA available whose underlying models are quite different. Distinguishing criteria are for instance the connection modeling, the incorporated knowledge and especially the level of abstraction. The main problem behind the existing approaches is the level of abstraction. Very detailed models will result in great modeling effort and consequently to high computational effort while using highly abstracted models will lead to incomplete results which means that some effects can't be revealed. Another problem of existing approaches is the integration into the development process. Most of the available analysis tools require models in special formalism independent of the models for simulation. Engineers will have to use a separate tool and spend much time in keeping the models consistent.

In this paper we present the modeling approach of smartIflow which was first described in [2]. It supports reasoning about failures and their effects through the whole product cycle. We briefly describe the concepts behind smartIflow and show how systems models can be composed graphically using Simscape and Stateflow.

This paper is organized as follows. Hereafter, we briefly analyze two existing approaches and their basic limitations. In Section 3 the concepts behind our modeling formalism are described. Graphical modeling will be explained in Section 4. In Section 5 the results of our experiments with an example system are shown. Finally a short conclusion will be given.

2. RELATED WORK

The MathWorks Simulink[1] and Simscape[2] are de facto standard for simulation of technical systems. In Simulink models are composed of a set of mathematical blocks such as integrators, derivatives or basic arithmetic operations that are connected via directed connections. The resulting block

[1]http://www.mathworks.com/products/simulink/
[2]http://www.mathworks.com/products/simscape/

diagram is equal to the mathematical model of the system under analysis. Simscape is an extension to Simulink, which enables physical modeling of multidomain systems. Models are composed of blocks representing the components in the real system. Simscape already provides a set of predefined blocks from various domains such as electric motors, resistors, or valves. Additionally, custom blocks can be created. The connections in Simscape are bidirectional. Thus the structure of the models matches the structure of the real system. Simulink and Simscape are optimal tools for simulation, but not for getting widespread information about system safety. The level of abstraction in particular of Simscape is too low. Both simulation tools use continuous solvers. Thus analyzing all possible failure situations would lead to state-space explosion. Furthermore, models in Simulink and Simscape only support deterministic behavior. To reach an appropriate level of completeness with acceptable computational efforts, one will often need more abstraction. The levels of abstraction of approaches to MBSA are quite different.

HiP-HOPS (Hierarchically Performed Hazard Origin & Propagation Studies) [6] is a modeling formalism using an extremely high level of abstraction. Models are based on Simulink block diagrams wherein a fault model is assigned to each block. The fault model is composed of a set of expressions which specify how a component responds to internal or failures created by other components. Fault modeling is quite limited since there are no state variables and failure propagation is restricted to one failure per connection. The directed connections also lead to the problem that situations in which the flow direction reverses can't be handled. In addition, the structure of both, the real system and the model is quite different. Multiple failure propagation channels can only be handled by creating multiple connections between components.

In contrast to HiP-HOPS, AltaRica [1] is a modeling formalism whose level of abstraction is somewhere between Simscape and HiP-HOPS. In principle, components in AltaRica are represented by nodes comprising variables, events, transitions, and equations to describe the behavior. Flow variables are used to exchange information between components, while state variables can be used to represent the operational- or failure-modes of a component. Events are used to represent state changes. Transitions consist of an event name that induces the state change, a guard and a list of state updates. A guard is the condition which must be fulfilled to enable a transition. As described in [5] there are well known limitations in the first AltaRica approach. AltaRica Data-Flow and AltaRica 3.0 [7] are the successor of AltaRica. AltaRica Data-Flow tries to reduce the computational effort by updating variables by using a fixed order for value propagation. However, no bidirectional flows and no looped systems are allowed. AltaRica 3.0 handles looped systems and bidirectional flows [7]. However, modeling physical flows is still very limited (there are no built-in elements for flow direction calculation) and events can only be triggered externally but not by components within system.

3. THE SMARTIFLOW FORMALISM

3.1 Basic Concept

In principle the main concepts behind AltaRica and smartIflow are quite similar. In fact, both approaches follow the principle of Discrete Event Systems (DES). SmartIflow regards components as finite state machines, which means that each component consists of a set of state variables and the corresponding values. In contrast to AltaRica, events that updates component states are generated based on signal changes initiated by other components. The behavior of a component is specified in terms of value changes of state variables as reaction on events, modifications of the network structure, and property publication through the network. Each component has a set of typed ports over which the components are linked. SmartIflow allows both directed and undirected connections. These connections are used to propagate abstracted physical flow information and additional data through the network. The propagated information is specified in terms of properties and depends on the current system state since the behavior of a component is state-dependent. Properties are key-value pairs and they are used to abstract from the physical values since we are not interested in specific signal values. Properties for instance can indicate a flow in a system that deviates from the expected value. Besides property propagation, the behavior description of a component also allows the modification of the network structure. For flow direction determination, smartIflow has some built-in components like sources, drains or bipolar sources. One simulation step is comprised of three sub steps. First, network structure and property values at specific flow variables are updated based on the current values of the state variables. Then, physical flow directions are calculated and property values are propagated through the network. Finally, the values of the state variables are updated based on events produced by changed property values.

3.2 Language

The following listing shows an excerpt of a valve modeled in the smartIflow modeling language.

```
class Valve {
   Ports:
      LogicalInput in1;
      Fluidal p1;
      Fluidal p2;
   States:
      Enum[Position1,Closed] s = Closed;
      Enum[StuckAtPosition1,Ok,StuckAtClosed]
                                 fm = Ok;
   Transitions:
      when(in1.val==Closed && fm==Ok &&
                s == Position1) {
         s = Closed;
      }
      when(in1.val==Position1 && fm==Ok &&
                   s == Closed) {
         s = Position1;
      }
   Behavior:
      if(fm==Ok && s==Position1) {
         connect(p1,p2,[flow.val=controlled]);
      }
      if(fm==StuckAtPosition1) {
         connect(p1,p2);
      }
      ...
}
```

The modeling language is component- and object-oriented. For each system component a separate class is created. Similar classes only need to be created once, because classes can be instantiated several times. Model components can, of course, be nested to create hierarchical structures. A component class comprises five *sections*, namely *Components*, *Ports*, *States*, *Transitions* and *Behavior*. The section *Components* is used to instantiate subcomponents. Each subcomponent can be accessed with a unique identifier. Connection points that are used to connect components are called ports. All ports of a component are declared in the section *Ports*. A port is specified by a type and identifier that must be unique within a component. The states and failure modes of a component are specified by state variables in section *States*. Each variable has a number of potential values and also a default value. The section *Transitions* describes how a component changes its state. Transitions are specified by a when-clause followed by a set of state assignments. Each when-clause consists of a logical expression that specifies the condition under which the state updates are performed. A logical expression is composed of state comparisons and comparisons between port properties. A state update is realized by assigning a new value to a state variable. In case of non-deterministic transitions, a number of new values, separated by the keyword *or* can be assigned. As already said, the behavior is state-depended. Therefore section *Behavior* consists of a set of conditional branches that specify the behavior in the different states. The conditions consist of a logical combination of state comparisons. The behavior of a component is specified by a set of actions namely *set* and *connect*. While *set* only puts a number of properties to a port, the *connect*-function links two ports and additionally a set of properties can be added. By omitting the if-clause, default actions can be created that are performed unconditionally.

4. GRAPHICAL MODELING

4.1 Motivation

Safety engineers are often not familiar with modeling languages such as AltaRica or smartIflow. Therefore creating system models in textual mode even with tool support is very complicated, tedious, time consuming and also error prone. Well known techniques from code editors like syntax highlighting, auto completion or code templates support the user in coding. However, it will be still hard to get an overview about the component instances and how they are interconnected. Another problem arises when specifying the transitions in smartIflow since the textual representation is very hard to understand in case of more complex components. A way to get rid of this problem is to use graphical modeling. Basically there are different ways to enable graphical composition for smartIflow models. One possibility could be for example to build a modeling tool from scratch or to use a framework like eclipse graphical modeling framework (GMF)[3]. Obviously this will lead to an optimal platform for modeling systems, however the engineers will then have to use another tool besides their standard simulation tools. Furthermore, this will result in additional effort since the models have to be created twice even though the structure of both models is quite similar. In addition, the engineers will have to spend a lot of time in keeping the models consistent. Since MatheWorks simulation tools Simulink and Simscape are widely used in industry, we decided to use them as platform for modeling technical systems in the smartIflow formalism.

[3]`http://www.eclipse.org/modeling/gmp/`

4.2 Matlab Integration

Since the language of smartIflow is component-oriented and object-oriented, we utilize the possibility of creating custom libraries in Simulink. This means that a component only needs to be modeled once and multiple instances can be created by drag and drop. Components are connected as usual. Figure 1 shows the structure of a valve modeled with the smartIflow formalism in Simulink.

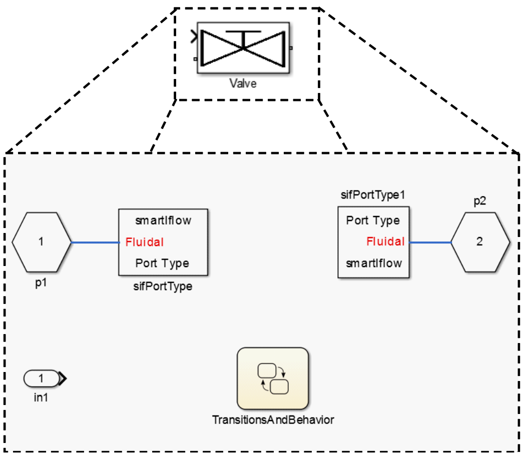

Figure 1: Component structure of a valve

Basically, each smartIflow class is represented by a Simulink subsystem. SmartIflow allows you to create hierarchical models. Thus each component can include several component instances. Ports are represented by Simulink *Inports* respectively *Outports* in case of directed connections and in case of undirected connections by Simscape physical ports (*PMIOPort*). In addition, a type must be specified for physical ports. This is realized by connecting a special component called *sifPortTyp* to a port. The type can be specified in the mask of the component. In case of our example component, there are two ports of type *fluidal* (*p1*, *p2*) and one input port (*in1*). When it comes to the representation of transitions and the behavior of a component, Stateflow seems to be a perfect candidate. Stateflow is a toolbox in Matlab for state diagram modeling. As shown in Figure 1, each component may contain exactly one Stateflow chart. This chart is used to represent the transitions and behavior of a component. As shown in figure 2, the Stateflow chart consists of two superstates, namely *Transitions* and *Behavior*. The state *Transitions* is used to define the state variables and, as the name implies, to specify the state transitions. Therefore, the substates of *Transitions* represent the various state variables of a component and their substates represent the corresponding potential values. Substates with dashed borders indicate a parallel decomposition, while solid borders indicate exclusive state decomposition. Default transitions (transitions with no source object) are used to specify the initial value of a state variable. Transitions between state values are labeled with conditions which specify, under which conditions the value change can take place. Non-deterministic transitions can be modeled by creating transitions that have the same source state and label and end at different destination states. The second superstate *Behavior* is used to define the state-dependent behavior. Each behavioral description is modeled by a sin-

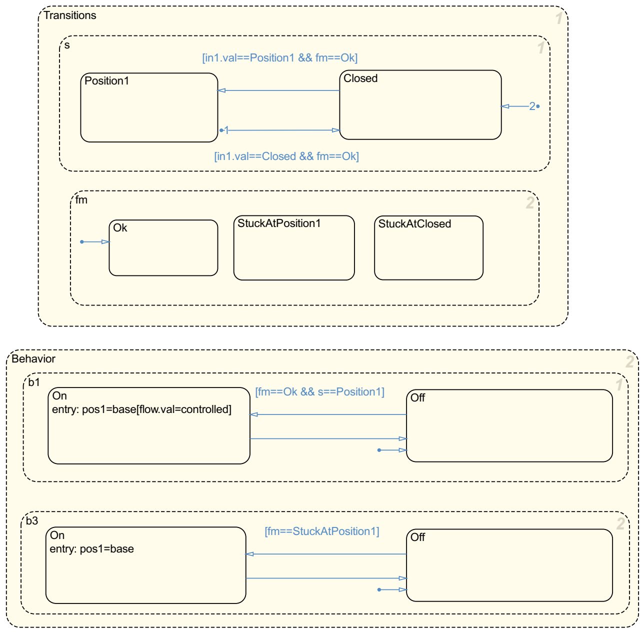

Figure 2: Transitions and behavior of a valve

gle state which contains two substates, namely *On* and *Off*. The transition from state *Off* to *On* specifies the condition under which the behavior is executed while the actual behavior is modeled as entry action inside state On. The behavior is described in a simplified syntax. For example, *pos1=base[flow.val=controlled]* connects the ports *pos1* and *base* and publishes the property *flow.val=controlled*. Whenever the condition of the Off-On-Transition is not fulfilled, the unconditional On-Off-transition will change the state of the behavior to *Off*. The modeling in Simulink has the same expressive power as the smartIflow language. Therefore, a system modeled with the smartIflow formalism in Simulink is semantically equal to the model in raw smartIflow classes. Only the kind of representation differs.

4.3 Conversion Algorithm

Since we need a clear interface between the graphical representation and the simulator, Simulink models are converted in the modeling language of smartIflow. Matlab provides a set of functions to access the blocks, their parameters and the structure of a Simulink model. The function *find_system* finds all blocks in a model and returns a handle to them. With function *get_param* and a block handle the properties of a block such as instance name, the name of the parent block or the port connectivity can be accessed. Stateflow charts can be accessed with function *find*. This function can be parameterized to analyze the composition of a chart and the transitions between the sates. Since smartIflow supports hierarchical models, the conversion algorithm starts at top level and goes recursively down to the blocks at the lowest level. All gathered information about a component is stored in an object, which is then finally written to a smartIflow class.

Our graphical conversion tool can be used to translate a Simulink model into raw smartIflow classes. The tool helps to select the input model and an output folder where the resulting classes are stored. To use the converter, it suffices to add the path to the conversion tool to the Matlab search path and enter *ConverterGUI* in the Matlab Command Window.

5. EXPERIMENTAL VALIDATION

Figure 3 shows an example system that we have successfully modeled in Simulink with the smartIflow formalism. The example shows a fuel system [8] that could be deployed in the real aircrafts. Main purpose of the system is to support the two engines with fuel that comes from the wing tanks. There are different ways of supporting the engines with fuel, such as crossover, where both engines are supplied by one tank (left/right). The model consists of over

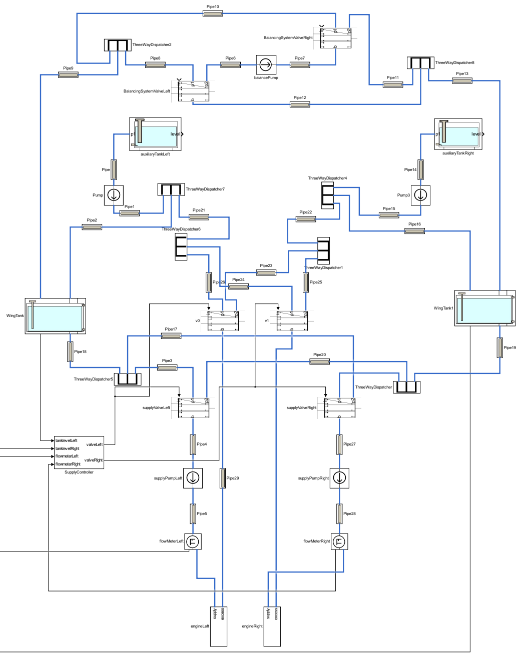

Figure 3: Example fuel system

60 components such as valves, pipes and controller. The conversion of the model into the smartIflow language took about 1.2 seconds on a workstation with an Intel Core i5 at 3.3 GHz running Windows 8.1 and Matlab 2014b.

Maucher discussed in [5] different ways of modeling the fuel system in AltaRica 1. The experiments have shown that the example system can be modeled only with massive simplifications since the formalism behind AltaRica has too many limitations (e.g. missing flow direction determination and incompatibility with looped systems). In addition, the lack of tool support for graphical model composition limits the practicality for bigger systems.

Thanks to the Simulink integration, modeling systems in HiP-HOPS is in principle quite easy and intuitive, even for more complex systems. The aircraft fuel systems consists of several pipes that allow flow in both directions. To handle such situations in HiP-HOPS, additional connections are necessary since HiP-HOPS uses unidirectional connections for information exchange between the components. The restrictions of HiP-HOPS (e.g. no state-dependent behavior) again lead to an extremely simplified system model.

6. CONCLUSION

In this work we showed the main concepts of smartIflow. SmartIflow tries to overcome the problems behind existing approaches, which mainly originates from the choice of the level of abstraction, by treating the connections between the components as information channels. This concept in combination with flow direction determination will help to handle situations with unforeseen flow direction changes. The Simulink integration allows even non-computer scientists to model systems quite easily. The component-oriented perspective of Simulink provides an optimal platform for modeling in smartIflow. Even complex transitions, can be specified straightforward in Stateflow. The experiments with the example system have shown that even large systems can be modeled in a simple and clear way.

7. REFERENCES

[1] 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.

[2] Philipp Hönig and Rüdiger Lunde. A new modeling approach for automated safety analysis based on information flows. In *25th International Workshop on Principles of Diagnosis (DX14)*, 2014.

[3] Anjali Joshi. *Behavioral Fault Modeling and Model Composition for Model-based Safety Analysis*. PhD thesis, Minneapolis, MN, USA, 2008. AAI3324436.

[4] J.C. Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 547–550, May 2002.

[5] Daniel Maucher. Model-based safety analysis with altarica.

[6] Yiannis Papadopoulos and JohnA. McDermid. Hierarchically performed hazard origin and propagation studies. In *Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 139–152. Springer Berlin Heidelberg, 1999.

[7] Tatiana Prosvirnova, Pierre-Antoine Brameret, and Antoine Rauzy. Model-based safety assessment: The altarica 3.0 project. *INSIGHT*, 16(4):24–25, 2013.

[8] Neal Andrew Snooke and Mark H. Lee. Qualitative order of magnitude energy-flow-based failure modes and effects analysis. *CoRR*, abs/1402.0581, 2014.