Formal Verification of Technical Systems using smartIflow and CTL

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

Verification of safety requirements is one important task during the development of safety critical systems. The increasing complexity of systems makes manual analysis almost impossible. This paper introduces a methodology for formal verification of technical systems with smartIflow. Existing approaches mainly use existing model checking tools for this task. Due to the bidirectional connection modeling, this is not feasible for smartIflow models. Our proposed two-step method first predicts the system behavior. In the second step, safety requirements specified in CTL are verified, and counterexamples are generated if these are not satisfied. We describe the usage of CTL formulas verified against safety critical complex systems modeled with smartIflow. The practical applicability is shown using a simple example system.

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, FSM, DES, Model Checking, CTL, LTL

1. INTRODUCTION

During development of safety critical systems serveral analysis tasks like FMEA (Failure Mode and Effects Analysis), FTA (Fault Tree Analysis) or CCA (Common Cause Analyse) are performed [5]. Besides that, safety engineers often verify the correctness of systems using safety requirements specifications. Performing this task manually can be time consuming and error prone since every system reaction to failures or external inputs have to be predicted. SmartIflow (State Machines for Automation of Reliability-related Tasks using Information FLOWs) [7] has been designed to automate the safety analysis process. The modeling formalism behind smartIflow is on a quite high level of abstraction. Due to the bidirectional connection modeling and the flexible property mechanism, the predictive power of smartIflow is quite high. Our previous experiments with smartIflow have shown that the level of abstraction offers a good compromise between computational effort and predictive power. However, in these experiments only specific scenarios (e.g. reaction to a broken pipe) were simulated. Existing approaches like AltaRica3 [3] or deviation models[11] use a similar level of abstraction, however, with less expressiveness (e.g. there are no built-in elements for flow direction determination).

This paper describes an approach for automated verification of technical system with smartIflow. The objective is to automatically verify a safety specification against a system model. In that context the following questions arise:

- How can such requirements be described in a formal language?
- How can the simulation be controlled to capture all relevant sequences of events?
- How might a verification result look like?

We will answer these questions by introducing our new verification method which is based on temporal formulas. It is being studied whether and how model checking techniques can be used.

This paper is organized as follows. The following section briefly summarizes the modeling concepts behind smartIflow. In Section 3 some fundamental model checking concepts are described. Existing approaches supporting formal verification of safety requirements are described briefly in Section 4. In Section 5 the automated verification of safety requirements with smartIflow will be explained. The results of our experiments with a small example system are shown in Section 6. Finally a short conclusion will be given.

2. THE SMARTIFLOW FORMALISM

The modeling formalism behind smartIflow is object- and component-oriented. Figure 1 visualizes the component model of a 3-position valve. Basically, each component in a system is considered as finite state machine. Thus a component consists of a set of state variables that are used to capture the operational and failure modes of a component. State changes are either performed after events or signal changes on ports initiated by other components. Events are used

ICAIT '16, Oct. 6 – 8, 2016, Aizu-Wakamatsu, Japan.

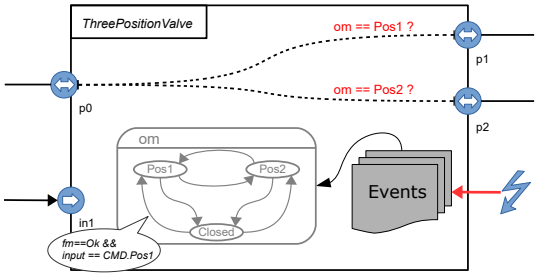

Figure 1: Sketch of a smartIflow component

to stimulate a system externally, for example to change the operational mode of a component. Ports represent the connection points of a component. There are two groups of port types, namely logical ports (e.g. input or output), and physical ports (e.g. for undirected electrical connections). Components are linked through these ports. The state-dependent behavior is described in terms of modification of the connection structure and also by property propagation through the network. Actually, properties are key-value pairs that provide a quite flexible mechanism to abstract from physical flow information. Connections are modeled as undirected channels in case of physical conductors, and there are also unidirectional channels for logical signals. Built-in primitives enable flow direction determination. The information about flow direction is available at each port by means of a reserved property (e.g. *flow.dir=IN*). Models in smarIflow can be composed graphically in Simulink/Simscape as we showed in our previous work [8].

3. MODEL CHECKING

Model Checking [2] describes the process of verifying a system model M against a specification ϕ. The objective is to proof whether M fulfills the specification ϕ. In other words: $M \vDash \phi$? This verification is performed fully automatically. System models describe the possible system behavior in a formal way. Finite state machines or transition systems are often used for behavioral description. The verification algorithm systematically explores all states and tries to disprove the specification. If the model doesn't fulfill the specification, the algorithm will deliver a counterexample, i.e. a trace of the system behavior that falsifies the property. However, most model checking tools like NuSMV deliver only one counterexample even though there could exist more [1].

3.1 Linear Temporal Logic

Most model checking tools allow specifications to be expressed in temporal logics, like LTL or CTL. LTL (Linear-time temporal logic) [9] allows to create formulas about future paths. Consequently, besides the usual logical operators ($\neg, \vee, \wedge, \Rightarrow$ *and* $\Leftrightarrow$) and operators for atomic expressions ($==, !=, <=, ...$) there is also a set of temporal Operators:

- $X\phi$: ϕ has to hold in the ne**X**t state
- $G\phi$: ϕ has to hold at each state of the subsequent path (**G**lobally)
- $F\phi$: ϕ has to hold somewhere on the subsequent path
- $\psi U\phi$: ψ has to hold **U**ntil ϕ holds
- $\psi R\phi$: ϕ has hold up to the moment when ψ becomes true (**R**elease)

An example for the safety specification "Every occurrence of an event is eventually followed by an action" expressed in LTL might look like as follows: G(event $\Rightarrow$ F(action)).

3.2 Computation Tree Logic

In contrast to LTL, CTL (Computation Tree Logic) allows to create statements about states of trees. Therefore CTL provides besides logical operators a set of temporal connectives. Each temporal connective is a pair of symbols. The first symbol specifies the path quantifier which is either A (along **A**ll paths) or E (there **E**xists at least one path). The second symbol stands for a set of temporal operators namely X (at the ne**X**t state), F (at a **F**uture state) and G (at all future states, i.e. **G**lobally). The combination of these two symbols defines the most important operators (EX, EF, EG, AX AF and AG) to specify properties that take into account the non-deterministic behavior of a system. Consider the following property: "It's always possible that state P occurs and holds for the rest of time". This may be expressed in CTL with the formula AF(EG(P)). Figure 2 shows two example systems that fulfills $AF\phi$ respectively $AG\phi$

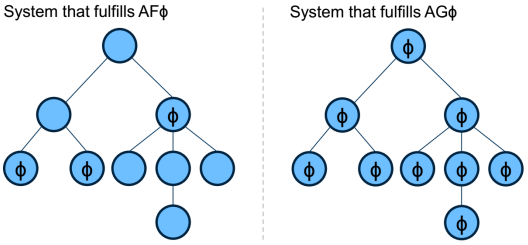

Figure 2: CTL: Examples

The expressiveness of LTL and CTL is quite different. There is an overlap, however, there are statements that can be expressed in CTL but not in LTL and vice versa. One should note that for instance NuSMV cannot generate counterexamples for all kind of CTL specifications [1].

4. RELATED WORK

Most approaches towards formal verification of safety requirements that can be found in literature utilize temporal logic and existing model checking tools. Joshi et al. proposes an approach based on Simulink and NuSMV model checker [10]. System models created in Simulink are extend with an fault model by using failure effect modeling. This so called extend system model is translated into the input language of the NuSMV model checker. In principle, Simulink is just used as alternative (graphical) representation for NuSMV models. Safety requirements specification created in CTL are used to verify the system behavior. If the system does not fulfill the specification, one counterexample with a trace of states that violated the specification is created.

The SLIM (System Level Integrated Modeling Language) language was developed by the COMPASS (Correctness, Modeling project and Performance of Aero space Systems) project for modeling hardware and software systems for safety-related tasks [4]. Their framework supports several analysis methods, among others, generation of FMEA tables, fault tree analysis, and correctness verification. Again, NuSMV is used as fundamental platform for the various analysis tasks. System models in SLIM are translated into the input lan-

guage of NuSMV and temporal logic is used for requirements specification.

Another approach based on model checking has been introduced by Güdemann et al. [6]. System models are constructed in SAML (Safety Analysis and Modeling Language) which can be translated in several analysis tools like NuSMV or PRISM. This approach allows both, qualitative and quantitative analysis.

5. FORMAL VERIFICATION OF SAFETY REQUIREMENTS

As described in the previous section, most approaches utilize existing model checking tools like NuSMV. This is obviously not feasible for smartIflow models, among other, due to the bidirectional connection modeling and flow direction determination. The input languages of existing model checkers don't provide such features, and because of that using existing model checker by translating smartIflow models in the input language of them is not possible. For this reason, we developed a model checking algorithm for smartIflow. Most model checking tools explore the system behavior depending on a requirements specification and try to disprove the specification. In our approach this is split up into two parts. First, the system behavior is simulated, and in the second step the specification is verified. This uncoupling has the great advantage that the verification is not dependent on the kind of simulation. Conversely, various specifications can be checked without re-simulating the system. However, this two-step approach may lead to problems when trying to keep search space small (e.g. only predict nominal behavior), because in that case we cannot profit from the specification. Both steps are now described in detail below.

5.1 Behavior Prediction

```
s = initial system state;
Q = ∅; // Open state nodes
Result = ∅; // Processed state nodes
add(Q,s) ;
while Q not empty do
    u = Q.remove() // Remove first item
    reconfigure network according to state(u);
    determine flows and propagate properties;
    v = compute possible next states;
    foreach possible next state e in v do
        if e not in Result then
            Q.add(e);
            Result.add(e);
        else
            create reference to existing state node;
        end
        if executeEvent(e) then
            next-states = execute admissible events;
            Q.addAll(next-states);
        end
    end
end
```

Algorithm 1: Behavior Prediction

Algorithm 1 describes the fundamental steps for behavior prediction. After a initialization phase, where all state variables are set to their default value, the subsequent system states can be predicted. A single simulation step basically consists of four substeps. During the **network reconfiguration**, connections are created and properties are published depending on the values of the state variables. After that, **flow directions are determined** and properties are propagated. Thereafter, **state variables are updated** depending on the current state variable values and port properties. In case of non-deterministic transitions, the result of this substep will be a set of subsequent system states. System states that already have been simulated previously get a reference to the existing state. Therefore, such states don't have to be simulated further, which reduces the computational effort, and also the total number of system states. Since we are interested in all possible system reactions on failures or input events, we have to stimulate the system using **events** at certain system states. We cannot execute events of a system in all possible combination at each state since this would lead to state-space explosion. This is also completely unnecessary, because there are a lot of constellations which can not occur in reality (e.g. several events at the same time). For this purpose we have developed a formal language that enables to describe the permissible combinations of events at a state node. For instance, events can be restricted to a special type or the number of events of a specific type (e.g. failure events) can be limited. Due to space limitations we cannot go into detail on the syntax and semantics of the language. Events are triggered only in certain states (e.g. stable state or alternating state). Therefore for each new expanded state node it is first checked whether events shall be triggered at all. After that, all possible events with respect to event specification are added to the state node and the possible subsequent states are determined.

A simulation result is a directed graph in which each node corresponds to one system state. Nodes reference other nodes, states do not reference states. Transitions which are caused by external events are labeled with a unique identifier of the event. Figure 3 shows a possible outcome of behavior prediction.

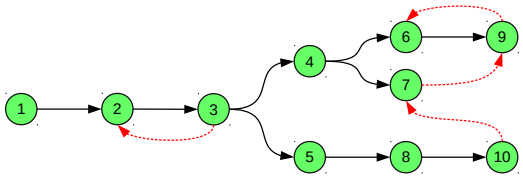

Figure 3: Simulation Result

5.2 Requirements Verification

The safety requirements need to be specified in a formal language. Typical safety requirements for a technical system look like as follows:

- "It is always possible to reach state X"
- "After pressing switch X must necessarily follow the action Y".

In case of the first requirement, LTL is not adequate since LTL can only express that X is actually reached and not that it can be reached. As already described in Section 3.1, LTL only allows to make specifications on a single path. CTL can state this property with formula $AG(EF(X))$. Therefore we decided to use CTL. Specifications can be created using all usual CTL operators ($AG, AF, AX, ...$). In atomic formulas, comparisons between all kind of variables including propagated properties and symbolic values can be expressed. Both, properties at ports and variables can be ac-

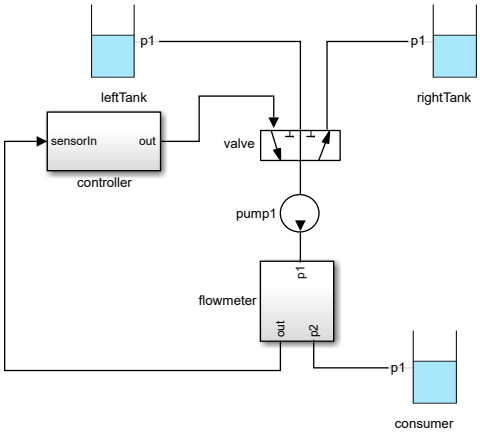

Figure 4: Example System

cessed via the absolute path, starting at root component (e.g. $MAIN.valve1.port1.flow.dir == IN$).

The evaluation of path quantifier in CTL formulas is in principle based on graph exploration. Depending on the operator, an expression ϕ is for instance verified at all paths at each node (in case of operator AG). ϕ can be any kind of expression, even an expression with a path quantifier. If ϕ is an expression with a path quantifier, the evaluation of the expression does not begin at root node of the simulation result, but rather at the current position of exploration. In case of a violation of a CTL formula, a counterexample is generated. A counterexample is characterized by a trace of events that have been executed on the path from the root node to the node where the specification is violated. Unlike NuSMV, which only generates one counterexample, our approach is able to create all counterexamples that violate a specification. However, there are still CTL expressions that will not return any counterexample. For example the expression $AG\phi \wedge AG\psi$ will just result to *TRUE* or *FALSE*. Generation of counterexamples is not possible in this case, since this would lead to ambiguities.

6. EXAMPLE

Figure 4 shows a very simple example system. The system consists of three tanks (left, right, consumer), a 3-position valve, flow meter, pump and a controller. The controller is responsible for keeping the consumer always supplied with liquid. In case of an empty tank (established through the flow meter), the controller is able to switch to the second tank. A safety requirements specification for this system may look like as follows: AG((((MAIN.leftTank.fm==Ok || MAIN.rightTank.fm==Ok) && MAIN.flowmeter.fm != Leakage) → AF(MAIN.consumer.p1.flow.dir == IN))). This means that at least one tank must be functional (no leakage, not empty), and there must be no leakage at flow meter, eventually the consumer is supported with fluid. Obviously this property is not satisfied. Consequently verification algorithm will deliver a quite comprehensive set of counterexamples. For instance the following event sequence will falsify the property: MAIN.v1.ActivateStuckAtPosition2 → MAIN.right.SetEmpty; The valve stuck at position two (supply from the right tank) and after that the right tank is empty. The controller tries to change the supply from the first tank, however the valve cannot changes the position. Therefore the consumer will no longer be supported with fluid.

7. CONCLUSION

In this work we described a method of formal verification of technical systems using smartIflow. Existing approaches to formal verification mostly use existing model checking tools. We cannot utilize these tools since the modeling formalism of smartIflow is quite different to input languages of existing model checkers. Despite the fact that we cannot use existing model checking tools, we use the powerful temporal logic CTL to specify the safety requirements. In contrast to model checking tools like NuSVM, our approach enables generation of multiple counterexamples. In the next step, we plan to enrich the events with information about the probability of failure to calculate the total probability of failure.

8. REFERENCES

[1] *NuSMV 2.5 User Manual*. Italy, 2010.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[3] M. Batteux, T. Prosvirnova, A. Rauzy, and L. Kloul. The altarica 3.0 project for model-based safety assessment. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 741–746, July 2013.

[4] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. *The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems*, pages 173–186. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[5] Federal Aviation Administration (FAA). *FAA System Safety Handbook, Chapter 9: Analysis Techniques*, Dezember 2000.

[6] M. Gudemann and F. Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pages 132–141, Nov 2010.

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

[8] P. Hönig, R. Lunde, and F. Holzapfel. Modeling technical systems with smartiflow for safety related tasks. In *International Workshop on Applications in Information Technology (IWAIT-2015)*, Aizu-Wakamatsu, Japan, 2015.

[9] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[10] A. Joshi, M. Whalen, and M. P. Heimdahl. Modelbased safety analysis: Final report. Technical report, University of Minnesota, 2005.

[11] P. Struss and S. 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.