[KRSW18] E. Kusmenko, B. Rumpe, I. Shumeiko, M. von Wenckstern:
Fast Simulation Preorder Algorithm.
In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pg. 256-267. SciTePress, Jan. 2018.
www.se-rwth.de/publications

Fast Simulation Preorder Algorithm

Evgeny Kusmenko, Bernhard Rumpe, Igor Shumeiko, Michael von Wenckstern ✉
Software Engineering, RWTH Aachen, Germany http://www.se-rwth.de
✉*vonwenckstern@se-rwth.de*

Keywords: Behavioral Compatibility, Component & Connector Model, Simulink, Evaluation on Automotive Models, Model-Checking

Abstract: Automotive industry uses model checking approaches to ensure behavioral backward compatibility of different variants and versions of software components to enable higher re-usability. Due to the lack of scalability, our already presented backward model-checking approach only allowed compatibility checks for small and mid-size components. Therefore, this paper presents several optimizations, such as normalizing and hashing the Expression Abstract Syntax Tree for faster evaluations and the creation of mappings for internal Simulink variables to avoid the need to unfold them. These optimizations lead to a tremendous decrease in execution time of our backward-compatibility checks between MATLAB Simulink components enabling the support of larger models. Besides describing the methodology behind the new fast simulation preorder algorithm, this paper also evaluates the different steps of the new algorithms for a driver assistant system provided by Daimler AG.

1 INTRODUCTION

Nowadays, embedded software controls the functional cycle of almost every device. Exploitation of such devices as cars is often bound with a risk for health and life of human beings. For that reason embedded software, used in automotive industry must be highly reliable. This means that significant amount of time and budget is dedicated by programmers to test and develop new software. Frequently, multiple versions of a software component or an extension of old software to use in a new context are required. This gives rise to the question, whether the new version can be used instead of the old one. In other words, if the components are backward compatible. To be backward-compatible, it must be checked if the new component contains the functionality of the old one. In this case, manual analysis is very error prone and time consuming and it is preferred to perform automated checks. In (Rumpe et al., 2015), *MontiMatcher*, a model-checking algorithm to automatically test Simulink components for behavioral equivalence was proposed. It is based on the idea of transforming Simulink components under consideration to input/output transition systems (I/O-TS) (Zhou and Kumar, 2012) and then to check whether all the execution paths of the second component are a superset of the execution paths of the first one. Thereby, a model checking tool, namely Microsoft's Z3 Solver, is used (Miller et al., 2010) to check whether guard conditions and/or output assignments are similar (you can ask Z3 is $\forall a,b,c \in [0,250] : c \cdot a + a \cdot b == a \cdot (b+c)$ and Z3 will answer *sat* (yes), *unsat* (no) or *timeout* (unknown)). It turned out that in some cases the execution of *MontiMatcher* can take more time than checking components for behavioral compatibility manually. For this reason this article presents a few optimization attempts aiming to reduce the time necessary to find out whether components are compatible.

The rest of the article content is given as follows: Section 3 provides more concrete information on the performed optimizations, Sections 4, 5, 6 describe disjunctive normal form, index tree, global variables optimizations respectively as well as a feasibility assessment of using the automated theorem prover Isabelle for behavioral compatibility checking (Paulson, 1994). Sections 7, 8, and 9 provide an evaluation, an overview of related works and a final conclusion.

2 FOUNDATIONS

Simulink (Mathworks Inc., 2016) is a component & connector based language very popular for model based software development in engineering disciplines, particularly in the automotive industry. In Simulink a component represents a piece of logical functionality. Input and output ports define the interface of such a component. Data flows between components is defined by connectors between output

ports (providing some data) and input ports (using this data). The behavior of a component may be provided either by MATLAB code or by a hierarchical decomposition into subcomponents. Furthermore, Simulink offers a large library of predefined components for multiple domains including control, signal processing, computer vision, and others.

This paper presents several optimizations for the model checking algorithm introduced by (Rumpe et al., 2015). The activity diagram for checking functional compatibility of two Simulink components is shown in Figure 1. If the components are compatible they can be replaced by each other in the system. Hence, it can be guaranteed that if one component is replaced by its newer version, the whole system continues functioning as expected, i.e., as with the old version of this component. It means that for all inputs that are accepted by the old version, the new version generates the same outputs. However, for inputs previously not accepted by the system, the new version may exhibit new behavior.

As the first step, the input Simulink models are transformed to a control flow graph. With its help all the output dependencies and update functions of internal variables are discovered. The next step is clone detection which is done on the basis of control flow analysis. It neutralizes all the parts of the component that do not have any influence neither on output calculation nor on internal variable updates. Such elements do not take part in the further analysis which simplifies the complexity. Then control flow graphs are transformed to Input/Output Extended Finite Automata (I/O-EFAs (Zhou and Kumar, 2012)) based on output ports. Thereafter, a state space calculation is performed which is basically a transformation of I/O-EFA to I/TO-TS where state names are combined with all possible values of internal variable update functions. This transformation often results in a state space explosion (Baier et al., 2008). As the next step the program locks all free ports allowing to take into account that new functionality of a component can be introduced by increasing the number of its input ports. The old and the new version can still be compatible if extra ports added to the new version are locked with some static values and so the new version exhibits the behavior of the old one. In the next step, a simulation is in fact running the algorithm described in detail by (Rumpe et al., 2015). The algorithm checks whether both I/O-TSs are in simulation preorder relation.

The next section states which problems in the program workflow we are tackling in this article and how we can solve them.

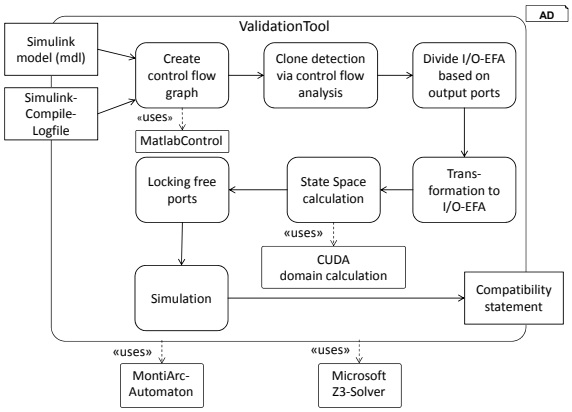

Figure 1: Program workflow to check compatibility of Simulink components (Rumpe et al., 2015).

3 MOTIVATION

As mentioned earlier, Simulink components are transformed to I/O-TS in order to perform behavioral compatibility analysis. The resulting I/O-TSs contain sets of states and transitions that represent execution paths of modeled software. The algorithm (presented in (Rumpe et al., 2015)) checks whether there is a simulation preorder relation between two I/O-TSs. Since the complexity of TSs is very high even for relatively simple components i.e they contain big number of states and transitions, checking for simulation preorder relation causes many SMT calls. These calls are needed to determine which outgoing transitions from a state of the second TS are activated if one transition from the current state of the first TS becomes active. To reduce SMT calls and consequently the execution time, the disjunction normal form (DNF) and index tree optimizations are introduced.

The DNF optimization organizes transitions guards such that semantically equal but syntactically possibly different guards obtain the same syntactical appearances. As a result repeated expressions that are joined via conjunction in guard conditions can be omitted making SMT calls more efficient. Furthermore, guard conditions can be compared syntactically avoiding SMT-calls and decreasing execution time. It is expected that this optimization will accelerate the checking procedure during the simulation step (see Figure 1) at the expense of reduction of SMT calls to check whether transitions of states of two I/Os are simulated by each other (or just in one direction).

To accelerate the selection process of implied transitions of an I/O-TS state, the index tree optimization was proposed. The idea is to build an index tree for each I/O-TS state of the first automaton to map guard

Figure 2: Steps needed to transform a Guard expression to optimized Hash-Guard-DNF expression.

variables to corresponding transitions. Then the index tree is queried with the guard variables of the second automaton to find all transitions of the first automaton implying the guard of the second automaton. One gets all transitions that can be potentially activated if the query transition becomes active. Consequently, an SMT solver does not have to check transitions that cannot be activated a priori. This leads to a higher efficiency of execution. This optimization is developed to reduce number of SMT calls even further and not to issue them for cases where it can be seen that the formulas to be checked (made of guard expressions) are a priori not satisfiable.

While two previously mentioned optimizations consider a possibility of acceleration of the transition filtering process when components are already transformed to I/O-TSs, the global variable optimization is intended to avoid transforming components to I/O-TSs (see Figure 1), since it leads to the state space explosion problem (Baier et al., 2008) even for simple components. The proposed optimization tries to find to correspondence between global variables of both components. If the matching is successful behavior compatibility check can be performed without unfolding of global variables. It means that in Figure 1 the step for transformation to I/O-TS is skipped and compatibility statement is directly derived without the simulation step.

The last optimization tries to formalize and solve the task of behavioral compatibility analysis in a generic theorem prover Isabelle. Performance gained is yet to be assessed. This article will discuss state of the research and show problems currently being faced.

4 DISJUNCTIVE NORMAL FORM OPTIMIZATION

The main purpose of the applied optimization is to normalize guard expressions to make syntactical and semantical comparisons equivalent. If such equivalence is reached, it allows to make statements on whether one guard expression is the same semantically without issuing an SMT-call. Reducing the number of SMT-calls in simulation preorder workflow makes the procedure consume less system resources and terminate in a shorter time.

The optimization sequence integrated in the Simulation-tool *MontiMatcher* is shown in Figure 2. The following sub-chapters explain each step of the Disjunctive Normal Form (DNF) transformation and provide examples for better understanding.

4.1 1st Step: Guard to Guard Normal Form (GNF) Transformation

At this step all the guard expressions are transformed into guard normal form. It is performed by equivalent transformation of guards to exclude the logical operators $<, >, \leq$ as well as the infix operator $-$. If possible distributive law of multiplication is applied and sums like $2 \cdot b$ are decomposed to summands: $b + b$. The right part of guards must contain zero. Let us take for example the following guard expression:

$$d \wedge (a \leq 10 \vee a \leq 10 \vee (b > 0 \wedge b > a) \vee \neg d) \quad (1)$$
$$\vee\, (3 + a) \cdot (c + 2 \cdot b) \geq b \quad (2)$$

Applying negation rule of boolean algebra, distributive law of multiplication and basic rules for inequality simplification we get:

$$d \wedge (10 + (-a) \geq 0 \vee 10 + (-a) \geq 0 \quad (3)$$
$$\vee\, (\neg(-b \geq 0) \wedge \neg(-b + a \geq 0)) \vee \neg d) \quad (4)$$
$$\vee\, (b + b + b + b + b + a \cdot c + c + c + c \quad (5)$$
$$+ a \cdot b + a \cdot b \geq 0) \quad (6)$$

After this step, algebraic expressions used in transition guards are normalized, e.g. they look the same syntactically if they have the same meaning. For example the expressions $b + a + a$ and $b + 2 \cdot a$ result in $a + a + b$, which enables syntactical comparison to check whether these are identical expressions.

The result of this step is the input for the 2nd transformation.

4.2 2nd Step: Guard-NF to Guard Disjunctive Normal Form (GDNF) Transformation

As it can be seen from the previous example, the resulting expression for the guard still contains conjunction operations. Applying the distributive law of boolean algebra to the last equations of the previous subsection we get an expression representing disjunction of conjunctive clauses:

$$[d \wedge 10 + (-a) \geq 0] \vee [d \wedge 10 + (-a) \geq 0] \vee$$
$$[d \wedge \neg(-b + a \geq 0) \wedge \neg((-b) \geq 0)] \vee [d \wedge \neg d] \vee$$
$$(b + b + b + b + b + a \cdot c + c + c + c + a \cdot b + a \cdot b \geq 0) \quad (7)$$

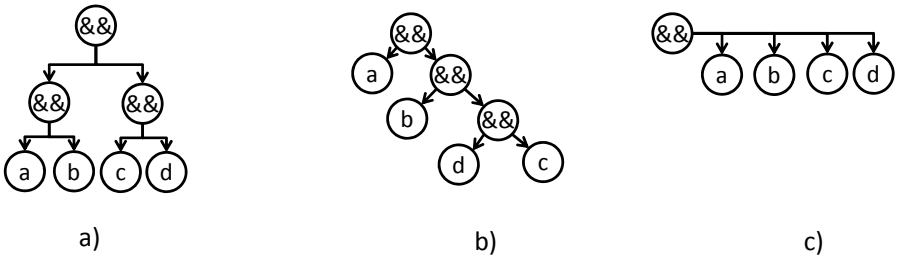

Figure 3: Prefix ASTs representation.

Having all the guard expressions in DNF allows to make statements whether one guard enables another one. For example, if both guard expressions have one or more equal conjunctive clauses it immediately means that they both evaluate to true or false if their variables them take the same value. Also if one of the guard conjunctors is known to be false or true it can be immediately concluded that the whole expression becomes false or true respectively. This accelerates the processes of making decision on satisfiability by the SMT solver.

The concatenation of guard conditions of the transitions of the two automata during the simulation process leads to many invalid guard expressions (such as $a \wedge \neg a$). The GDNF step detects these invalid conditions, and therefore, these conditions will not be processed later anymore.

4.3 3rd Step: Guard-DNF to Hash-Guard Disjunctive Normal Form (HGDNF) Transformation

At this step new hash ASTs (Abstract Syntax Trees) were introduced to store and perform operations on GDNF expressions. One of the operations allowing to speed up comparison of two ASTs is the ability of such objects to produce hash values that correspond to their inner structure. Hash values are used to make it possible to sort summands in guards, e.g that the expressions $b+v+2\cdot a+c$ and $v+c+a+b+a$ have the form $a+a+b+c+v$ after the transformation. To normalize ASTs, the prefix notation (as shown in Figure 3 (c)) is used to join the same operation nodes wherever possible into a single operation node. Therefore, conjunctions like $(a \wedge (b \wedge (c \wedge b))))$ are normalized and can be compared syntactically. This was not possible before since two structurally different AST expressions would produce different hash values, even though they are the same semantically like in Figure 3. Applying Prefix-ASTs reduces also the time needed for traversing the nodes of the tree ; when traversing the infix tree in Figure 3 a) and b) the algorithm needs to visit 7 nodes while in c) it only needs 5 nodes. The expression shown in Equation 7 can be represented as an AST in prefix form:

$$\begin{aligned}
&\vee(\geq(+(b,b,b,b,b,b,c,c,c,\cdot(a,b),\cdot(a,b),\cdot(a,c)),0)\\
&\quad\wedge(\geq(+(-b,a),0),(d),\neg(\geq(-b,0)))\\
&\quad\wedge(\neg(d),(d))\\
&\quad\wedge(\geq(+(-a,10),0),(d))\\
&\quad\wedge(\geq(+(-a,10),0),(d)))
\end{aligned} \tag{8}$$

Figure 4 shows a part of the AST hash tree for the example. Pay attention that all the nodes are sorted in accordance with the hash order.

4.4 4th Step: HGDNF to Optimized HGDNF Transformation

To further simplify the AST expressions we can make use of Hash ASTs to detect situations in which the guard conditions can by directly evaluated to true of false.

$$\#(\vee a,b) = \#_{java}(\vee\_a\_b) = \#(\vee b,a) \tag{9}$$
$$\#(<a,b) = -(\#(\geq a,b)) \tag{10}$$

To achieve this, the ASTs participating in the guard condition must be compared with each other. In case of equal hash values an additional comparison of ASTs must be performed, since a hash function is non-surjective, and therefore applying it to different arguments can lead to the same result. If the AST check shows a positive result, it can be concluded after the SMT check (due to non-surjectivity of the used hash function) that both ASTs are equal. But in cases where the hash values of ASTs under comparison are non-equal the AST check can be skipped and the procedure demonstrates higher performance since the AST check takes considerably more time than the comparison of two integer values. Thus, since $\#(\neg(d)) = -\#((d))$ the respective disjunction can be evaluated to false and excluded after the AST

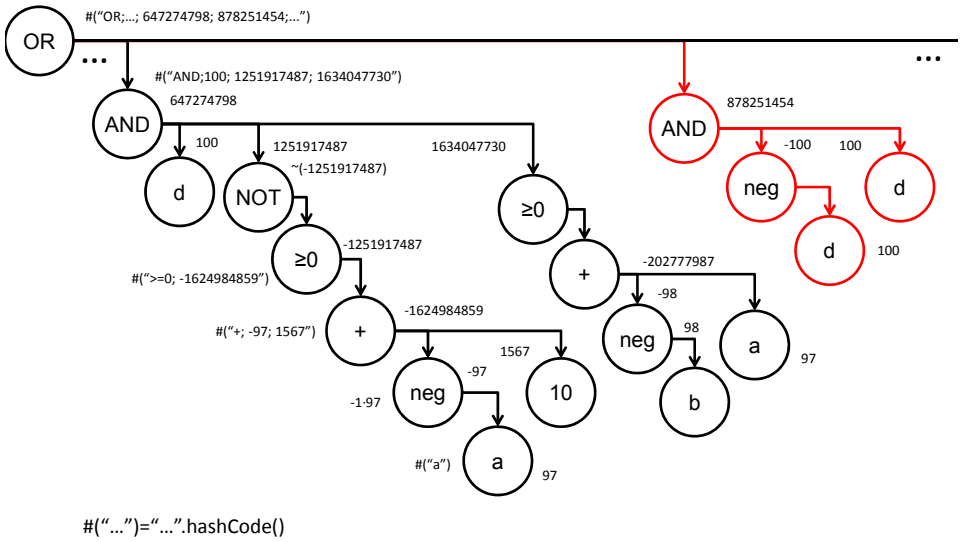

Figure 4: Part of the AST tree for Equation 8.

check. As a result we get:

$$\begin{aligned}&\vee(\wedge(\geq(+(-b,a),0),(d),\neg(\geq(-b,0)))\\&\quad\geq(+(b,b,b,b,b,b,c,c,c,\cdot(a,b),\cdot(a,b),\cdot(a,c)),0)\\&\quad\wedge(\geq(+(-a,10),0),(d))\\&\quad\wedge(\geq(+(-a,10),0),(d)))\end{aligned}\tag{11}$$

After this transformation step it becomes possible to extract port variables from the guard expressions. A set of guard variables is used in the next section in order to build the index tree for transition filtering optimization.

5 INDEX TREE OPTIMIZATION FOR TRANSITION FILTERING

One step in the automata simulation preorder relation algorithm (Rumpe et al., 2015) checks which transitions of the second I/O-TS can potentially be activated by a given transition of the first I/O-TS. This optimization excludes transitions from the set of potentially activated transitions; the smaller set size results in less expensive Z3 calls to filter the actually activated transitions from the potential ones. For example in Figure 5 to understand whether the transition C in Figure 5a is simulated by the transitions of A_S(i) in Figure 5b, each set of port variables of the guard expression in GDNF form (i.e. {a, b} and {c}) of transition C must be checked for being a super set for sets of conjunction variables for transitions of A_S(i).

5.1 Earlier approach

Using AST expressions allows to extract all the ports' variables used on a given transition. The algorithm checks whether the variable set of the AST guard expression of the given transition ($tr1$) is a superset of variables of a selected transition taken from the other automaton. If it is the case, the first transition ($tr1$) potentially activates the one ($tr2$) from the other automaton. To exclude wrong implications such as $\neg a \Rightarrow a$ where the left guard is superset of the right guard, a SMT call checks the complete implication.

More formally: $A1$ and $A2$ are variable-sets containing all used variables in $Guard1$ and $Guard2$. Guards are defined as conjunction of expressions with only one non-trivial input variable ($term_v, v \in Inputvariables$, $Interpretation(Dom(v)) \not\models term_v$): $Guard1, Guard2 \in \bigwedge_i term_v$. Then, the following holds: $(A1 \not\subseteq A2) \Longrightarrow (Guard2 \not\Rightarrow Guard1) \Longrightarrow (Guard1 \not\sim Guard2)$. From $A1 \not\subseteq A2$ we know that $A1$ contains at least one element that is not in $A2$. Example: $a \wedge b \not\Rightarrow a \wedge c$, since $\{a,b\} \not\supseteq \{a,c\}$.

After port variable extraction, the Java and Guava collection frameworks check the superset relation between variable sets. For that for transition of A_S(i) the procedure must go through all the port variables sets for transitions of B_S(i). Obviously it is not very efficient since all the transitions of B_S(i) are checked numerous times and in general during the simulation process states of I/O-TSs can be visited several times, and for each time the filtering procedure must be executed with the same overhead.

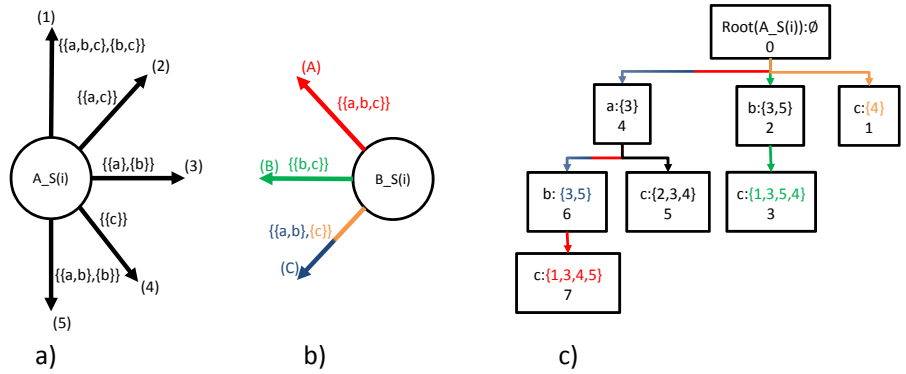

Figure 5: To help answer the question whether state B_S(i) (b) simulates state A_S(i) (a), the index tree (c) was built.

5.2 Index tree approach

The bottle neck of the described approach is that the algorithm must iterate through all the transitions of A_S(i) for each transition of B_S(i). Thus, in our example all the A_S(i) transitions must be traversed three times. Taking into consideration that during the simulation check states of I/O-TSs can be visited several times and, therefore, this procedure may be repeated, this can create an overhead in terms that the same operations are executed numerous times.

As an attempt to make this procedure faster an index tree (as shown in Figure 5 c) is created. Its size depends on the number of port variables participating in transitions of A_S(i). The conjunction subsets of B_S(i) serve as queries for the tree. Numbers of the activated transitions of A_S(i) are obtained as an answer. Such implementation is possible since all the variables of the conjunctions are sorted and if further traversing of the tree becomes not possible, it states that the answer to the query can already be read. Current implementation uses a pre-calculated structure to reference nodes that contain transitions activated by the current traversing path.

For example if the node number 6 ($\{a,b\}$) has been reached it means that the query activates all the transitions referenced as index for nodes 2 (a only) , 4 (b only), and 6 (a and b). The variables a, b, and c are only place holders; and since the variables are ordered they can be mapped to them directly and the predefined structure with their dependency links (as in the example $6 \rightarrow \{2,4,6\}$) can be used. It allows to traverse all the transitions of A_S(i) only once. This traverse happens during creation of the index tree. After the index tree is built transition filtering is done just by querying the tree, which is estimated to be faster then going though all the transitions each time. The procedure only has to create the index tree once which allows to save time if the I/O-TS states are revisited.

6 GLOBAL VARIABLE OPTIMIZATIONS

6.1 Strategy to avoid internal variable unfolding by using Z3

Now we demonstrate a strategy for the minimization of global variable unfolding. It is possible if dependencies between internal variables can be found.

Since Microsoft's Z3 solver cannot find arbitrary dependencies (complex mapping functions from one variable to another one) automatically, this algorithm focuses on linear dependencies between two variables as it can ask the Z3 solver to calculate (if they exist) the coefficients (factor and offset) for the linear mapping from one variable to another. In case, the automata contain non-linear parts in the output statement and/or update functions, it may be possible to find linear relationship between some variables. The remaining ones must be unfolded to a I/O-TS.

Two components shown in Figure 6 are taken as an example. They have different structures but execution yields the same output for both automata if the input is equal. Buffers (denoted as Z^{-1}) define global variables. The values calculated in the previous time step are used for the computation. The initial values of the internal variables are indicated as input to the I/O-automata, that are the models of the shown components. Input ports are denoted as a,d,e for the first automaton and as a',e',d' for the second one. Operations performed on numbers are summation and multiplication.

As the first step all the linear relationships are discovered by requesting SMT solver to check the internal variables on linear dependency. An example of such a check is shown in Equation 12. The equation expresses that there is a correspondence between global variables of two components. If this is the case for some variables SMT solver defines at this step

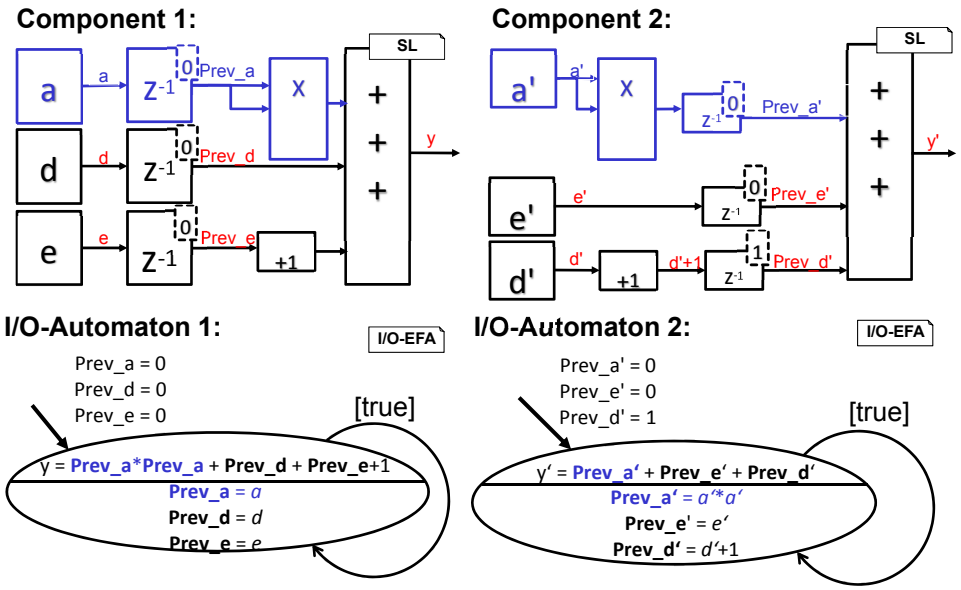

Figure 6: Components and their respective automatons for comparison.

which equations can be solved against coefficients x. Apparently the expression $(x11Prev\_a + x12Prev\_d + x13Prev\_e + x10) = a \cdot a$ cannot be resolved since it contains non-linear operation $a \cdot a$. As the result of this step this expression is discarded as well as coefficients $x11, x12, x13, x10$. It means that there is non-linear relation between any of the global variables of the first I/O-automaton and the variable $Prev\_a'$ of the second automaton.

$$\begin{aligned}
&\forall d, e, Prev_a, Prev_d, Prev_e \\
&(Prev_a = a \wedge Prev_d = d \wedge Prev_e = e \\
&\wedge (x11Prev_a + x12Prev_d + x13Prev_e + x10) = a \cdot a \\
&\wedge (x21Prev_a + x22Prev_d + x23Prev_e + x20) = e \\
&\wedge (x31Prev_a + x32Prev_d + x33Prev_e + x30) = \\
&d + 1)
\end{aligned} \tag{12}$$

After removing the non-linear part of the system in Equation 12, it becomes Equation 13. Invoking the SMT solver on Equation 13 finds all linear coefficients $(x20 = 1, x21 = 0, x22 = 1, x23 = 0, x30 = 0, x31 = 0, x32 = 0)$ to express relationship between the variables of the automaton.

$$\begin{aligned}
&\forall d, e, Prev_a, Prev_d, Prev_e \\
&(Prev_a = a \wedge Prev_d = d \wedge Prev_e = e \\
&\wedge (x21Prev_a + x22Prev_d + x23Prev_e + x20) = e \\
&\wedge (x31Prev_a + x32Prev_d + x33Prev_e + x30) = \\
&d + 1)
\end{aligned} \tag{13}$$

Finally the expressions for the output functions are formulated considering calculated coefficients. Both automata are unfolded into the both I/O-TSs shown in Figure 7, whereby only the remaining variables $Prev\_a$ and $Prev\_a'$ are unfolded. This significantly reduces the number of states in the I/O-TSs. If the linear correspondence between the variables of both automata were not detected, all three global variables would need to be unfolded and this would result in an I/O-TS with 27 states and 729 transitions for both automata. Comparing these two automata with three unfolded variables would lead to 19 683 SMT calls in the simulation phase only to check whether transitions of the first automaton implies transitions of the second one. In contrast to that, the new approach (which removes all linear-dependent variables between both automata) needs only 27 SMT calls. Already on this simple example we gain a speed-up factor of over 700 for the simulation step. Applying the simulation algorithm of the automata in Figure 7 reveals that both systems produce the same output for the same input.

Due to the control-based behavior (e.g PID controllers) of cyber-physical systems many previously calculated values (e.g. for calculating differences as approximations of derivatives) need to be stored. Most likely, if two systems have the same behavior, both systems also store the same previously calculated values; but since Simulink derives internal names for atomic blocks based on the graphical layout of the system, the names of the blocks that store the same values may be different. This approach was created to address this issue.

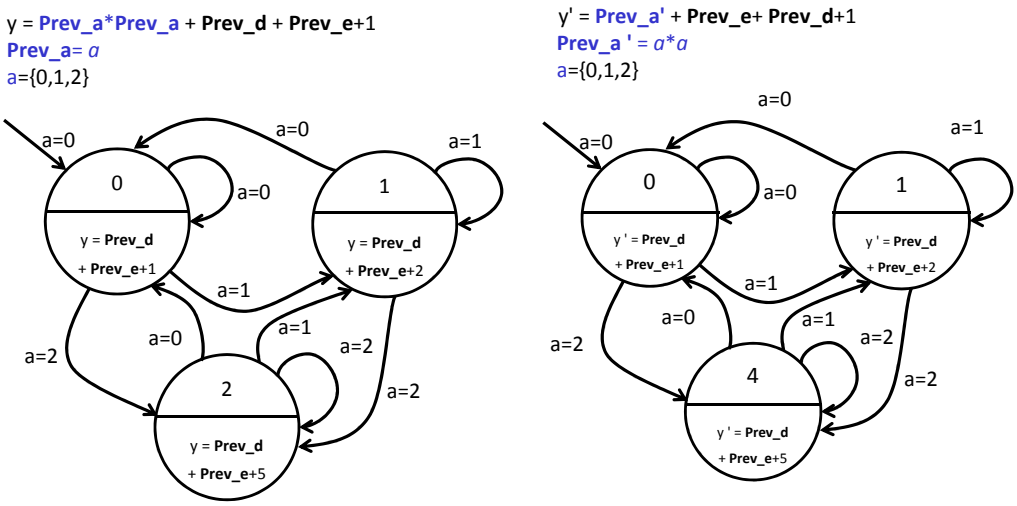

Figure 7: The I/O-TSs for the components in the example. It can be seen that both I/O-TSs have the same execution sequences.

6.2 Strategy to avoid internal variable unfolding by using Isabelle

An alternative direction of research is to check behavioral compatibility using an automated theorem prover. Figure 8 demonstrates how compatibility of the components in the example can be checked with the Isabelle theorem prover (Paulson, 1994). The update functions are defined with definitions and then both outputs are shown to be equal. However for more complex components this approach has some problems with automatization of the compatibility check, since update functions are often recursive and equality of output functions must be proved by induction. Induction proves frequently require defining several supplementary lemmas that describe dependency between global variables update functions. Finding dependencies of such relations is hard to automatize.

Isabelle

```
definition Prev_a :: "nat⇒(nat⇒nat) ⇒ nat" where
"Prev_a t a=(if t=0 then 0 else a (t-1))"

definition Prev_aprime :: "nat⇒(nat⇒nat) ⇒ nat" where
"Prev_aprime t a=(if t=0 then 0 else (a (t-1))*(a (t-1)))"

definition Prev_d :: "nat⇒(nat⇒nat) ⇒ nat" where
"Prev_d t d=(if t=0 then 0 else (d (t-1)))"

definition Prev_dprime :: "nat⇒(nat⇒nat) ⇒ nat" where
"Prev_dprime t d=(if t=0 then 1 else (d (t-1))+1)"

definition Prev_e :: "nat⇒(nat⇒nat) ⇒ nat" where
"Prev_e t e=(if t=0 then 0 else e (t-1))"

definition Prev_eprime :: "nat⇒(nat⇒nat) ⇒ nat" where
"Prev_eprime t e=(if t=0 then 0 else e (t-1))"

lemma presentation_example :
"Prev_a t a*Prev_a t  a+Prev_d t d+Prev_e t e+1=
Prev_aprime t a+Prev_eprime t e+Prev_dprime t d"
 using Prev_a_def Prev_aprime_def
       Prev_d_def Prev_dprime_def Prev_e_def Prev_eprime_def
       by auto
```

Figure 8: Isabelle proof to show that components are equal.

7 EVALUATION

The optimized *MontiMatcher* framework using the here presented fast simulation preorder algorithm has been tested against the Advanced Driver Assistent System (ADAS) provided by Daimler AG. In a previously case study (Bertram et al., 2017) on structural requirement verification we made all Simulink models via web-export available under: `http://www.se-rwth.de/materials/cncviewscasestudy/`.

In the rest of this evaluation section we use the following abbreviations:

CC4 CruiseControl component of version 4

CC3 CruiseControl component of version 3

FLS FAS Limiter component with sign detection

FLN FAS Limiter component without sign detection

_act active output port of FLS or FLN

_kmh kmh (speed) output port of FLS or FLN

_lim *LimiterSetVariable* in Simulink subsystem of FLS or FLN, this is reduced to an output port as all components of the ADAS only need the current value and no history.

Performance improvements gained via the DNF optimization are given in Table 2. In cases when both tested components are almost equal the optimization leads to up to 47 % acceleration for the overall process. For non-compatible components or simulation directions the optimized procedure shows some loss

```
definition prev_a :: "nat⇒(nat⇒bool)⇒ bool" where                Isabelle
"prev_a t a=a t"

definition out1 ::"nat⇒(nat⇒bool)⇒bool" where
"out1 t a =(if(prev_a t a ∧ a t) then True else
if(¬prev_a t a∧a t) then False else
if(prev_a t a∧¬a t) then False else True )"


definition prev_a2 :: "nat⇒(nat⇒bool)⇒ bool" where
"prev_a2 t a=a t"

definition out2 ::"nat⇒(nat⇒bool)⇒(nat⇒bool)⇒bool"
 where "out2 t a c =(if(prev_a2 t a ∧ a t ∧ c t) then
   True else if(¬prev_a2 t a∧a t ∧ c t) then False else
if(prev_a2 t a∧¬a t ∧ c t) then False else
if (¬c t) then True else True )"


lemma  portFixation :"∃a2. out1 t  b1  = out2 t  a1 a2"
 using out1_def out2_def prev_a_def prev_a2_def by auto
```

Figure 9: Port fixing Isabelle prove. The left right hand part contains b1(t) function instead of a1(t). Despite that the output functions are proved to be equal

of performance, since in this case more information needs to be disproved. Besides that, processes to build supplementary constructions inside the automata objects also take computation resources. Such processes include transforming all nodes into AST hash-nodes, computing DNF expressions, concatenating, etc. It is also important to notice that if the automata are reused in later simulation checks, the performance gain will be greater, since all the necessary supplementary constructions are built on the previous steps.

The simulation results of the global variable optimization are shown in Table 1. It can be seen that in comparison to the DNF results the algorithm demonstrates **acceleration up to ten times** (FLS lim ← FLN lim). Also for some tests we translated Z3 queries to the Isabelle theorem prover language. It was done to check whether using Isabelle can bring more performance gain. From the results in Table 1, it can be concluded that for most of the conducted test usage of the theorem prover does not lead to a shorter execution time. However we continue research in this direction. For the cases where Isabelle results mentioned as *unsat*, decision on compatibility is made by an initial check, which finds that ports ranges are different and there is no necessity to issue an SMT call to state that the components are not compatible. In the case of *measurement not taken* the theorem prover was not able to handle the request, most probably because of its size.

The dependency of time needed to perform transition filtering from number of transitions in a state is shown in Table 3. In comparison to the Java Collections Framework the presented approach allows to get up to 50 % execution time gain. But unfortunately this acceleration is not enough to cover the expenses paid for the index tree building. The simple procedure based on classes taken from the Java Collections Framework performs fast enough due to well optimized methods of the used classes. Though the index tree approach brings some benefits, it is discarded and must be reconsidered in course of the future research.

Table 1: Simulation Results using GVT. The corresponding best results from Table 2 are the denoted by the first number in the second column; the numbers in brackets denote the old values from (Rumpe et al., 2015).

Simulation Results (GVT)		
test #	Time Z3 (ms)	Time Isabelle (ms)
CC4 → CC3	251(4766)	326
FLS_cc→FLN_cc	124(359)	61
FLS_lim→FLN_lim	251(3156)	499
FLS_act→FLN_act	109(390)	855
FLS_kmh→FLN_kmh	798(3364)	measurement not taken
CC4 ← CC3	47(0)	unsat
FLS_cc←FLN_cc	109(375)	39
FLS_lim←FLN_lim	188(3047)	519
FLS_act←FLN_act	94(391)	860
FLS_kmh←FLN_kmh	139(0)	unsat

Table 2: Global Variable Test evaluation.

	None Opt.		DNF Opt.		Improvment %
test #	time	SMTs	time	SMTs	
CC4(r.) → CC3(r.)	734	9	515	7	29,84
CC4(r.) ← CC3(r.)	0	0	16	0	0
CC1(r.) → CC3(r.)	2081	16	2298	30	-10,43
CC1(r.) ← CC3(r.)	2950	33	2271	29	23,02
CC1 → CC3	1749	20	3364	43	-92,34
CC1 ←CC3	3831	48	3035	40	20,78
CC4 →CC3	7156	83	5156	62	27,95
CC4 ← CC3	0	0	79	0	0
FLS_kmh(r.) →FLN_kmh(r.)	187	3	188	3	-0,53
FLS_kmh(r.) ← FLN_kmh(r.)	0	0	0	0	0
FLS_kmh → FLN_kmh	0	0	78	0	0
FLS_kmh ← FLN_kmh	6711	81	3594	49	46,45
FLS_cc → FLN_cc	422	6	422	6	0
FLS_cc ←FLN_cc	406	6	422	6	-3,94
FLS_lim → FLN_lim	3250	48	3156	48	2,89
FLS_lim ← FLN_lim	3125	48	3047	48	2,5
FLS act → FLN act	531	8	407	6	23,35
FLS act ← FLN act	578	8	422	6	26,99

8 RELATED WORK

Model checking is a convenient, reliable and automatic method to test whether a software is compliant with the predefined requirements (Reinbacher et al., 2008) (Zhang et al., 2010). As mentioned in earlier sections, this article is based on the work by

Table 3: Execution time before and after the DNF optimization is applied.

time, ms			
transition #	non opt.	tree building	indexed trans. filtering
10	0	264	0
20	20	9740	10
35	37	17693	20
40	78	31452	40
50	141	78694	100

(Rumpe et al., 2015) wich describes a model checking approach for compatibility verification of Simulink components. Bounded model checking is used to conduct behavior compatibility analysis of ANSI-C program and circuits in Verilog. Software component behavior can be transfered into π calculus expressions to check for compatibility based on behavior descriptions (Zhang, 2009). Labeled transition systems are used for compatibility checking for asynchronously communicating software (Ouederni et al., 2013). Group transition systems and ABS modeling language are used to perform model-based compatibility analysis of software after modifications (Poetzsch-Heffter et al., 2012).

(Chakrabarti et al., 2002) shows a methodology of interface compatibility checking for software modules. The given approach is to use reachability analysis algorithms on pushdown games. Another approach for checking behavioral compatibility of interfaces is given in (Wang and Krishnan, 2006). The work describes developing a Simple Component Interface Language (SCIL) that is derived from interface automata. The language allows users to perform compatibility analysis without having a strong mathematical background required to use formal methods.

An approach to checking behavioral compatibility between web services was described in (CHAE et al., 2008), it employs an extended version of the conventional methods rule and state machines. Pertri nets are also used to analyze behavioral similarity of web services (Li et al., 2011). Open bisimulation, modal mu-calculus and π calculus are used for formal verification of web services (Gao et al., 2006).

In (Dragomir et al., 2016) compatibility analysis of Simulink components was performed with the Isabelle theorem prover. The work presents a tool set that translates Simulink models into Isabelle theories. Simulink models are represented as predicate transformers and not as function like in our work. Such a method has the advantage that it can model components that can fail for particular input values, for example division by zero.

Note that clone detection is a similar task to behavior compatibility checking as a cloned component would exhibit the same behavior whereas two components with the same behavior should be detected as clones.

In (Alalfi et al., 2012b) three types of model clones are defined:

- Type-1: exact model clones
- Type-2: structurally identical model clones, except for variation in labeling (i.e. renamed blocks) and values types.
- Type-3: model parts (fragments) with changes in connection or block position, as well as small amount of additions and removals of blocks. Block labels and value types can differ (i.e. they are renamed).

In (Liang et al., 2014) another definition is used to distinguish between the types of model clones:

1. Syntactic model clones:
   - Exact model clones: structurally the same models
   - Approximate model clones: syntactically similar models, with slight difference in labels, values or attribute variations, but essentially the same structure.
2. Semantic model clones: models that have the same behavior, but rather different structures.

For clone detection several approaches exist in the literature: In model driven software development, which is quite often the preferable way of designing software systems in the automotive field, a cloned block might result in using more components while an automobile is getting build, which also means that the building cost of the product might increase too. In (Rumpe et al., 2015) a discussion about version compatibility and maintenance in Simulink models reveals problems that occur in automotive systems and how clone detection improves optimization algorithms. Most of the methods and algorithms used to find clones in software systems aim to find structural matches, i.e., syntactic clones. As an example we can point out (Pham et al., 2009) - a graph-based clone detection tool for Matlab/Simulink models, detecting both exactly matched and approximate model clones, (Nguyen et al., 2009) - a structural characteristic feature extraction tool, (Pham et al., 2009) - a graph-based clone detection tool for Matlab/Simulink models, detecting both exactly matched and approximate model clones, (Jürgens et al., 2009) - a framework for clone detection, allowing ungapped and gapped clone detection, using suffix-tree, generated by the program unit sequence.

Clone detection is also used for solving other problems, for example in (von Detten and Becker, 2011) an approach combining clustering and pattern-based detection of mistakes in component-based software implementations, in (Abi-Antoun et al., 2006) an algorithm for comparing and merging C & C architectural views, in (Wille et al., 2013) an approach to analyze related models and determine the variability between them using structural clone detection, or in (Stephan and Cordy, 2015) a near-miss cross-clone detection technique, used to find anti-patterns in Simulink models.

While there are many implementations adopting the structural clone detection approach and providing good results, the number of false positives found by those methods is still large. In order to increase precision and find meaningful clones, behavioral clone detection methods have been developed. As an example for such tools and algorithms we can point out (Alalfi et al., 2014) - a plugin called Simone, detecting and representing variability in Simulink models based on text-based clone detection, (Antony et al., 2013) - a tool called NiCad, detecting near-miss clones in UML behavioral models, by using text-based approach over XML, (Alalfi et al., 2012a) - near-miss clone detection, based on transformation of graph-based models to normalized text form, using Simone plugin as extension to NiCad tool, (Stephan et al., 2013a) - another paper, describing detection of model clones in Simulink using Simone plugin to track the evolution of model clones with respect to their clone containing classes, (Deissenboeck et al., 2010) - a clone detection tool based on an industrial case study undertaken with BMW Group, using graph-theory technique, (Stephan et al., 2013b) - a model-clone detection framework, based on mutation-analysis, using graph theory, (Stephan et al., 2012) - a brief review over Simulink model clone detection approaches. Behavioral clone detection mostly relies on structural clone detection tools and algorithms.

9 CONCLUSION

This paper presented several optimization methods to accelerate compatibility checks of software components applied in the automotive field. The conducted experiments indicated that the optimization undertaken to normalize transition guards of I/O-TS to Disjunctive Guard-Normal-Form and then to Optimized Hash-Guard-Disjunctive-Normal-Form results in a decent performance gain of the overall behavioral compatibility checking process. Therefore, this optimization has been included into the *MontiMatcher* framework. Additionally, this paper compared the performance when using the automated theorem prover Isabelle as a replacement for Microsoft's SMT-Solver. We also show how to generate Isabelle code to perform compatibility checks and assessed how Isabelle performed against Z3 solver for these kinds of application. Though it is still possible to use Isabelle, the measurements show that the SMT-Solver Z3 is in most cases faster. Therefore, it is necessary to search for another appropriate candidate for the replacement or to endure speed losses, that in many cases can be critical. We also tried and evaluated an index tree as a transition filtering mechanism. Unfortunately, while querying the tree shows a better performance than the previously used simpler approach, the construction process of the tree appeared to be very slow due to a large number of access operations during the construction. The optimization removing internal variables improved, as expected, the speed of the compatibility checks for mid-scale component and connector models a lot. In cases when this internal variable optimization cannot make any statement on compatibility, it reports fast enough and, therefore, can be used before the whole execution chain of the old *MontiMatcher* tool is invoked.

Acknowledgements Special thanks goes to the two students Vladimir Parashin and Igor Shumeiko who implemented these optimizations in their bachelor and master theses supervised by Michael von Wenckstern.

REFERENCES

Abi-Antoun, M., Aldrich, J., Nahas, N. H., Schmerl, B. R., and Garlan, D. (2006). Differencing and Merging of Architectural Views. In *ASE*.

Alalfi, M. H., Cordy, J. R., Dean, T. R., Stephan, M., and Stevenson, A. (2012a). Models are code too: Near-miss clone detection for Simulink models. In *ICSM*.

Alalfi, M. H., Cordy, J. R., Dean, T. R., Stephan, M., and Stevenson, A. (2012b). Near-miss model clone detection for Simulink models. In Cordy, J. R., Inoue, K., Koschke, R., Krinke, J., and Roy, C. K., editors, *IWSC*.

Alalfi, M. H., Rapos, E. J., Stevenson, A., Stephan, M., Dean, T. R., and Cordy, J. R. (2014). Semi-automatic Identification and Representation of Subsystem Variability in Simulink Models. In *ICSME*.

Antony, E. P., Alalfi, M. H., and Cordy, J. R. (2013). An approach to clone detection in behavioural models. In Lämmel, R., Oliveto, R., and Robbes, R., editors, *WCRE*.

Baier, C., Katoen, J.-P., and Larsen, K. G. (2008). *Principles of model checking*. MIT press.

Bertram, V., Maoz, S., Ringert, J. O., Rumpe, B., and von Wenckstern, M. (2017). Case Study on Struc-

tural Views for Component and Connector Models. In *MODELS*.

CHAE, H. S., LEE, J.-S., and BAE, J. (2008). An Approach to Checking Behavioral Ccompatiblity Between Web Services. *IJSEKE*, 18(02).

Chakrabarti, A., de Alfaro, L., Henzinger, T. A., Jurdziński, M., and Mang, F. Y. C. (2002). *Interface Compatibility Checking for Software Modules*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Deissenboeck, F., Hummel, B., Jürgens, E., Pfaehler, M., and Schätz, B. (2010). Model clone detection in practice. In Inoue, K., Jarzabek, S., Koschke, R., and Cordy, J. R., editors, *IWSC*.

Dragomir, I., Preoteasa, V., and Tripakis, S. (2016). Compositional semantics and analysis of hierarchical block diagrams. In *International Symposium on Model Checking Software*. Springer.

Gao, C., Liu, R., Song, Y., and Chen, H. (2006). A Model Checking Tool Embedded into Services Composition Environment. In *GCC*.

Jürgens, E., Deissenboeck, F., and Hummel, B. (2009). CloneDetective - A workbench for clone detection research. In *ICSE*.

Li, X., Fan, Y., Sheng, Q. Z., Maamar, Z., and Zhu, H. (2011). A petri net approach to analyzing behavioral compatibility and similarity of web services. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 41(3).

Liang, Z., Cheng, Y., and Chen, J. (2014). A novel optimized path-based algorithm for model clone detection. *JSW*, 9(7).

Mathworks Inc. (2016). Simulink User's Guide. Technical Report R2016b, MATLAB & SIMULINK.

Miller, S. P., Whalen, M. W., and Cofer, D. D. (2010). Software model checking takes off. *Commun. ACM*.

Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. (2009). Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In Chechik, M. and Wirsing, M., editors, *FASE*, volume 5503 of *LNCS*.

Ouederni, M., Salaün, G., and Bultan, T. (2013). Compatibility checking for asynchronously communicating software. In *FACS*.

Paulson, L. C. (1994). *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media.

Pham, N. H., Nguyen, H. A., Nguyen, T. T., Al-Kofahi, J. M., and Nguyen, T. N. (2009). Complete and accurate clone detection in graph-based models. In *ICSE*.

Poetzsch-Heffter, A., Feller, C., Kurnia, I. W., and Welsch, Y. (2012). *Model-Based Compatibility Checking of System Modifications*.

Reinbacher, T., Kramer, M., Horauer, M., and Schlich, B. (2008). Motivating Model Checking of Embedded Systems Software. In *MESA*.

Rumpe, B., Schulze, C., Wenckstern, M. v., Ringert, J. O., and Manhart, P. (2015). Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *SPLC*. ACM New York.

Stephan, M., Alalfi, M. H., Cordy, J. R., and Stevenson, A. (2013a). Evolution of Model Clones in Simulink. In Pierantonio, A. and Schätz, B., editors, *Workshop on Models and Evolution*, volume 1090 of *CEUR Workshop Proceedings*.

Stephan, M., Alalfi, M. H., Stevenson, A., and Cordy, J. R. (2012). Towards qualitative comparison of Simulink model clone detection approaches. In Cordy, J. R., Inoue, K., Koschke, R., Krinke, J., and Roy, C. K., editors, *IWSC*.

Stephan, M., Alalfi, M. H., Stevenson, A., and Cordy, J. R. (2013b). Using mutation analysis for a model-clone detector comparison framework. In Notkin, D., Cheng, B. H. C., and Pohl, K., editors, *ICSE*.

Stephan, M. and Cordy, J. R. (2015). Identification of Simulink model antipattern instances using model clone detection. In Lethbridge, T., Cabot, J., and Egyed, A., editors, *MODELS*.

von Detten, M. and Becker, S. (2011). Combining clustering and pattern detection for the reengineering of component-based software systems. In Crnkovic, I., Stafford, J. A., Petriu, D. C., Happe, J., and Inverardi, P., editors, *ISARCS*.

Wang, L. and Krishnan, P. (2006). A framework for checking behavioral compatibility for component selection. In *ASWEC*.

Wille, D., Holthusen, S., Schulze, S., and Schaefer, I. (2013). Interface variability in family model mining. In *SPLC*.

Zhang, C. (2009). Software components composition compatibility checking based on behavior description. In *GRC*.

Zhang, P., Muccini, H., and Li, B. (2010). A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5).

Zhou, C. and Kumar, R. (2012). Semantic translation of simulink diagrams to input/output extended finite automata. *Discrete Event Dynamic Systems*, 22(2).