Improving Invariant Mining via Static Analysis

CHRISTOPH SCHULZE and RANCE CLEAVELAND, University of Maryland

This paper proposes the use of static analysis to improve the generation of invariants from test data extracted from Simulink models. Previous work has shown the utility of such automatically generated invariants as a means for updating and completing system specifications; they also are useful as a means of understanding model behavior. This work shows how the scalability and accuracy of the data mining process can be dramatically improved by using information from data/control flow analysis to reduce the search space of the invariant mining and to eliminate false positives. Comparative evaluations of the process show that the improvements significantly reduce execution time and memory consumption, thereby supporting the analysis of more complex models, while also improving the accuracy of the generated invariants.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Software reverse engineering**; Documentation;

Additional Key Words and Phrases: Invariant mining, automated test generation, model-based development, verification and validation

ACM Reference format:
Christoph Schulze and Rance Cleaveland. 2017. Improving Invariant Mining via Static Analysis. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 167 (September 2017), 20 pages.
https://doi.org/10.1145/3126504

1 INTRODUCTION

Accurate and complete information about the actual behavior of a software system is essential to developers and users of the system. Developers need such information to make informed decisions during development, maintenance, and evolution; users need similar information to interact with the system properly and safely. Unfortunately, traditional manually maintained system specification documents are often incomplete, inaccurate, or out-of-date, and they can impede the development as well as the effective and safe usage of the system after it has been deployed.

To counteract this problem of misleading specifications, researchers have proposed the use of *invariant mining* on run-time system artifacts [1, 5, 8]. The intuition is that these specifications can convey information about system behavior at a much higher level than system code, while being free from the inaccuracies that often plague developer-maintained specifications. These techniques have been shown to yield useful information about systems, and even to reveal flaws in system specifications. However, the underlying data-mining algorithms used by these tools typically are computationally complex, and this introduces efficiency problems when the techniques are applied

Research supported by NSF Grants #1446583, #1446365, #1446665, as well as the Fraunhofer Center for Experimental Software Engineering.
This article was presented in the International Conference on Embedded Software 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: C. Schulze and R. Cleaveland, University of Maryland, 8223 Paint Branch Dr, College Park, 20740; emails: cschulze@umd.edu, rance@cs.umd.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 ACM 1539-9087/2017/09-ART167 $15.00
https://doi.org/10.1145/3126504

to larger artifacts. In addition, as artifacts grow in size the likelihood of *spurious invariants* also increases, thereby degrading the accuracy of the generated information.

This paper shows how the problems of scalability and accuracy may be addressed using static analysis. The particular artifacts we focus on are Simulink®[1] models; such models are widely used in industries such as automotive as a basis for the generation of embedded code. We show that by adding simple data- and control flow analyses into our invariant-generation technique in [1] we can dramatically improve both the efficiency of invariant mining and the accuracy of the invariants that are produced.

The remainder of this paper is organized as follows. The next two sections review our invariant-mining technique for Simulink in [1] and discuss issues that arise when it is applied to larger models. The section following then describes our static analyses and how they are intended to address the scalability and accuracy problems in [1]. We then develop an experimental framework for evaluating the new techniques using a corpus of 12 Simulink models of automotive and medical-device control systems, and demonstrate statistically how the static analyses lead to faster run times, lower memory usage, and better invariants. The final sections discuss threats to the validity of our work, related work, conclusions and directions for future work.

2 BACKGROUND

This section reviews Simulink, our existing invariant-extraction framework for Simulink [1] and the invariant mining algorithm that we employ to extract invariants.

2.1 Simulink

Simulink is a graphical block-diagram modeling language developed by The MathWorks Inc. Simulink is used in conjunction with The MathWorks' MATLAB® tool; Simulink models may also contain submodels given as hierarchical state machines in the Stateflow® hierarchical notation.[2]

Because of its ability to model both discrete- and continuous-time dynamics, Simulink is widely used in the automotive, ground transportation and aerospace industries to design control systems. Discrete Simulink models are also often used as specifications for embedded control software, and tools such as The MathWorks' Embedded Coder and dSpace's TargetLink can automatically generate deployable C code from these models.

2.2 Invariants via Testing and Data Mining

Our invariant-generation approach [1] combines automated testing with *association-rule mining* [3] to create invariants from Simulink models. The general techniques follows an iterative *test* -> *infer* -> *instrument* -> *retest* cycle, and is depicted in Figure 1.

- **Generate Test Data.** Test data is automatically generated from system models using the Reactis®[3] automated coverage testing tool. Reactis actively strives to cover large portions of the systems behavior by targeting different structural coverage metrics, including decision coverage and modified condition/decision coverage (MC/DC), as well as Simulink-specific coverage metrics such as conditional-subsystem coverage.
- **Infer Invariants.** Invariants are association rules, which take the form of implications whose left-hand sides (LHSs) refer to inputs and internal variables and whose right-hand sides (RHSs) are outputs of the system (e.g. $input1{=}1 \wedge internal1{=}1 \Rightarrow output1{=}1$). These outputs may themselves be new values for the internal state variables. The invariants are

[1]Simulink® is a registered trademark of The MathWorks, Inc.
[2]MATLAB® and Stateflow® are registered trademarks of The MathWorks, Inc.
[3]Reactis® is a registered trademark of Reactive Systems, Inc.

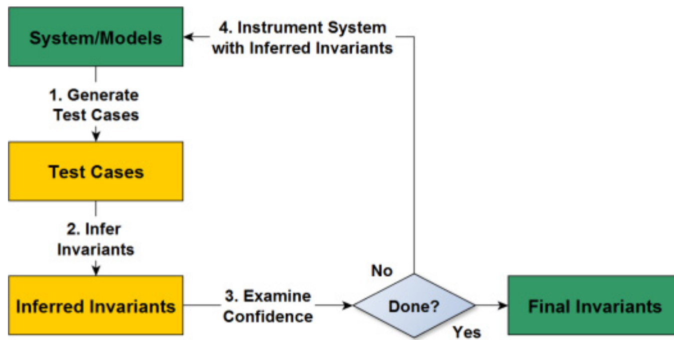

Fig. 1. Overview of the specification extraction approach.

inferred from the test data generated in the previous step using a modified Frequent Pattern (FP)-Growth [13] algorithm from the Sequential Pattern Mining Framework (SPMF) [10][4].

- **Instrument System.** To validate the proposed invariants, the system models are instrumented with so-called *monitor models* that represent the proposed invariants. As their name suggests, these monitor models observe the underlying system for any violations of the associated invariants. Reactis provides support for automating this instrumentation processes. (In languages such as C/C++ this could be achieved via assert statements.)
- **Retest.** The purpose of the retesting phase is to check whether the inferred invariants can be falsified with additional testing. Since the invariants are attached to the models as observers, they are treated as additional coverage objectives by Reactis, which now actively tries to find counterexamples to the proposed invariants. In the process it either disproves invariants, or strengthens the confidence in them by creating additional test data that supports them. In addition to validating existing invariants the additional test data can also uncover new behaviors which lead to invariants that were missed in previous iterations. The test data of all previous iterations is aggregated with the newly created test data and a new set of invariants is inferred in each iteration. If the monitor model detected a counterexample, then the corresponding invariant does not hold true for all test data and is automatically discarded by the data miner. Thus, iterating the approach leads to a more accurate set of invariants.
- **Terminating the Process.** The process terminates if there is no change in the set of invariants for N iterations, where N is a parameter set by the user.

This framework is designed to infer invariants of Simulink models, but it can be adapted for other types of systems, provided the following requirements are met: *Automated test generation* technology needs to be available; *Observability* of the inputs, internal states, and output values that should be in invariants must be possible; *Monitoring* of the invariants during system execution, without changing system behavior, must be possible.

Experiments have shown [1] that test generation technologies that actively try to invalidate the proposed invariants tend to produce more accurate invariants. Furthermore, some of the optimizations introduced in this paper rely on the availability of data- and control flow information between the variables.

[4]The Magnum Opus tool used originally in [1] is no longer available.

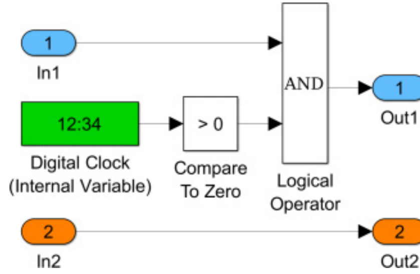

Fig. 2. Example model illustrating a constellation that leads to spurious invariants.

2.3 Association Rule Mining via FP-Growth

In order to mine association rules from the test data, we employ the FP-Growth algorithm [13]. The algorithm works in two steps.

In the first step it identifies frequently co-occurring sets of *items*. Consider {*in1=1, in2=1, out1=1*}, this is a set of 3 items or 3-item set. Each item set has a support value, that indicates how often the items appear together in the data. For example if the item set had a support of 5 there would be five co-occurrences of in1=1, in1=1 and out1=1 in the test data. In order to identify this item set the algorithm first identifies all frequent single items by counting the support of each item in the test data. If the item occurs frequently enough (based on a user defined threshold) the item is retained, otherwise it is eliminated. The algorithm then identifies the frequently occurring 2-item sets, 3-item sets to n-item sets (n in our case is bounded by the number of inputs, internal variables and outputs). For information on how the FP-growth algorithm identifies these item-sets efficiently we refer the reader to [13].

In the second step the algorithm then creates association rules by taking one item from the set to be the RHS of the rule, with the remaining items in the set forming the LHS of the rule (e.g. $in1{=}1 \wedge in2{=}1 \Rightarrow out1{=}1$). The algorithm then calculates how often the rule is true, which is the so called confidence of the rule. If the confidence passes a given threshold (1.0 in our case, which means there are no counterexamples) the rule is accepted.

3 SCALABILITY AND SPURIOUS INVARIANTS

A pilot study in [1] highlighted the utility of the invariant-generation approach discussed in the previous section: missing pieces in a specification of production automotive control software were uncovered. Other experiences with the tool similarly produced valuable insights into model behavior. However, in experiments we undertook on a variety of other automotive models we encountered some limitations of the basic framework. Two of these issues—*Spurious Invariants* and *Explosion of the Search Space*—became the motivation for this work. We describe these below.

3.1 Spurious Invariants

While the retest cycle in [1] identifies many false positives, there are cases where the approach returns an objectively true invariant (one that cannot be disproven by the data) that gives misleading information about the behavior of the system. One of the reasons why this happens is that the association-rule mining process only identifies statistical relationships, not causal dependencies, between variables. As a side effect there are cases where the approach identifies invariants between variables that have no actual causal dependency in the system itself; we refer to these invariants as *spurious*. Figure 2 contains a simplified Simulink model that can give rise to spurious invariants. The system has two inputs, two outputs and an internal clock. The clock is used for the

initialization step of the system and ensures that Out1 is zero after the system has booted up. Note that in our example input in2 is not connected to output out1; this is the intended functionality based on the requirements of the system. In these types of scenarios the approach infers spurious invariants relating values of in1 and out2, even though no connection exists between them. When spurious invariants are possible because of inputs that are unconnected to outputs in this fashion, they can overwhelm the actual invariants; in our experiments we observed ratios of actual invariants to spurious invariants as high as 1:100 depending on the number of in and outputs of the system. These spurious invariants make the analysis of the results much harder for the user and slow down the test generator considerably since it must process a larger number of invariants.

3.2 Explosion of the Search Space

This problem occurs in larger models that have many inputs, internal variables, and outputs. It leads to usage of large amounts of time and memory, and in our experiments even crashed due to lack of memory on the 32GB machine used during our evaluation.

To understand the source of the problem, consider the FP-Growth algorithm that we introduced in Section 2.3. In order to create association rules it has to find all possible frequent item sets in the data, leading to a potentially very large search space which is exponential to the number of unique items in the data. The general complexity of this problem is NP-hard but for special cases (e.g. bounded minimum support) it has been shown to be in P [4]. To generate the item sets the algorithm creates several tree structures called *FP-Trees*; the number of trees is also proportional to the number of unique items in the test data. So the main driver of the performance (execution time and memory consumption) of the algorithm is the number of unique items that it has to consider. The frequent items in our data are based on the inputs, internal variables and outputs of the system and their associated values from the test data, which means that with each additional variable that we have to consider the unique items increases by the number of values that this variable can take.

Another factor increasing the number of item sets that we have to consider is the *support* setting. The pruning of items that do not have enough support can reduce the search space greatly. However, since we want to identify all invariants no matter how rare they are, we have to use a very low minimum support threshold. Our evaluation shows that there is a big trade-off between performance vs accuracy. One can greatly speed up the data-mining process and reduce memory usage, but this is at the expense of missing invariants. Furthermore, without a low support threshold there can be cases where the approach does not converge. It turns out that some invariants might fall below the support threshold in one iteration, only to later reappear again due to the way the test data changed between iterations. If the new test data in an iteration lifted the support value for a previously dropped invariant over the threshold again it would reappear. This fluctuation of invariants shows that, besides not finding all invariants, a support setting that is too high can lead to non-convergence of our approach, since the accuracy of the set of invariants may not be guaranteed to increase monotonically with the number of iterations.

3.3 Other Limitations

In addition to the above-mentioned shortcomings the approach only works with *categorical variables*. The regular association-rule-mining algorithm treats each numerical value of an integer or floating point variable as a separate category. For example, it would create the invariants $speed=24.4 \Rightarrow mode=off$, $speed=24.45 \Rightarrow mode=off$ and so on. The validation step would then try to find counterexamples to these rules creating more test data and the subsequent data mining step would create even more invariants. This stops the approach from converging since it always can find more invariants in each step. For this paper, we address this problem via user-supplied data abstractions. Specifically, the user can supply information about data abstraction that the approach

will apply before the data mining process is invoked. We do not address this further in this paper, except in the future work section.

4 BETTER INVARIANTS VIA STATIC ANALYSIS

In this section we show how to address the problems of Spurious Invariants and the Explosion of the Search Space using data- and control flow dependency information. This information will allow us to eliminate spurious invariants and reduce the search space of the data mining process.

We examine two different uses of this information. The first approach introduces constraints in the FP-Growth algorithm as described in [21]. This leads to fewer recursive calls, thereby reducing the search space. Furthermore, all invariants that are produced this way only entail variables with actual causal dependencies, which will remove spurious invariants. The second approach partitions the test case data in such a way that for each output variable there is one dataset that only entails the inputs and internal variables that have a causal dependency to that output. The data-mining algorithm is then executed separately for each of these datasets. This also reduces the search space and removes the spurious invariants.

In the rest of this section we describe the static analyses and how the information is factored into the data-mining process.

4.1 Dependency Analysis

Our approach to pruning the search space of the invariant mining process and identifying spurious invariants involves determining which inputs and internal variables can affect the values of which outputs of a system. We compute this information by building a directed *dependency graph* from the Simulink model. Each input into the system, each read of an internal system variable, and each system output will be represented as a distinct vertex in this graph; computing causal information for a given output just involves performing a reachability calculation on this graph (e.g. using depth- or breadth-first search). The transformations discussed in this section are specific to Simulink models. However, the graph representation can be reused for other data-flow languages for which such dependencies can be computed. The rest of this section describes how this dependency graph is constructed for Simulink models.

Simulink models consist of *blocks*. Each block contains a collection of inputs and outputs together with logic describing how the block executes. Roughly speaking, the model of execution for a block is as follows. Inputs to the block are read when they are available, after which the block "fires" by executing its logic, thereby producing values on each of its outputs. Outputs of a block may in turn be connected to inputs of another block, inducing as a result a dependency between the blocks: the downstream block's outputs will depend in part on the outputs produced by the upstream block.

Simulink blocks may take different forms. So-called *basic blocks* are the smallest building blocks of Simulink models. Many such blocks, such as the Sum block, compute functions over their inputs, yielding results as outputs. Other blocks, such as Zero-Order Hold, have state that depends on previous values provided as inputs. Similarly, Data Store Read blocks permit internal variables created via Data Store Memory blocks to be read, while Data Store Write blocks allows these variables to be modified. (In this paper, for simplicity we assume that any value written to a Data Store is also output on some Outport. This simplifies our account of invariants as having "outputs" on the RHS.) Still other blocks are called *virtual*, as they are primarily responsible for routing data through a model. Examples of the latter include Inport and Outport blocks.

Other blocks take the form of *subsystems*, which contain submodels. The inputs and outputs of a subsystem correspond to the top-level Inport and Outport blocks of its submodel. In addition, certain subsystems also have special triggering inputs indicating when the model should, or should

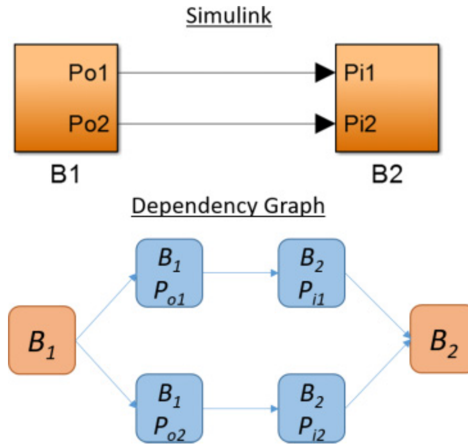

Fig. 3. Transforming a Simulink block into a dependency graph.

not, execute. In this regard, while Simulink is primarily a data-flow language, there are also control-flow aspects to its operational semantics.

In addition to basic blocks and subsystems, Simulink also includes blocks that may be used to include C code, or MATLAB code, or state machines in the Stateflow notation, within models. In this paper we will not consider these blocks.

4.1.1 Dependency Graph Construction. Our goal is to build a directed graph whose vertices are basic blocks as well as their outputs and inputs, and whose edges reflect data-flow / control-flow dependencies. We describe this construction in stages, beginning with generic basic-block assemblages and then continuing with our treatment of certain specialized blocks, including subsystems. Given space constraints, this discussion is necessarily abbreviated.

4.1.2 General Block Treatment. Dataflow between two basic Simulink blocks B_1 and B_2 is treated as in Figure 3. In this case, we introduce a vertices for B_1 and B_2 into our dependency graph, together with vertices for each output and input of the block. Outgoing edges in the figure connect B_1 to its outputs, while incoming edges connect B_2 to its inputs. Finally, any connections in the Simulink diagram between outputs of on block and inputs of another are added as directed edges in the obvious fashion.

4.1.3 Virtual Blocks and Subsystems. Our dependency graph is intended to be a flattened representation showing causal connections between basic non-virtual blocks, as well as top-level Inports and Outports. The description below illustrates how we "translate away" subsystems and certain virtual blocks. We focus in particular on the following four cases: *From* and *GoTo* blocks; *Virtual Subsystems*; *Conditional Subsystems*; and *If Else* blocks. Graphical depictions of the corresponding transformations are given in Figure 4. Other types of virtual blocks and subsystems are treated similarly and are omitted.

GoTo/From. *From* and *GoTo* blocks (Example 1 in Figure 4) introduce virtual connections between blocks. The GoTo block GB_1 and the From blocks FB_1 and FB_2 in the example contain the tag *[A]*; this indicates that there are connections between the output of block B_1 and the inputs of B_2 and B_3. Our construction adds vertices for the GoTo/From blocks and directed edges from GoTo blocks to corresponding From blocks.

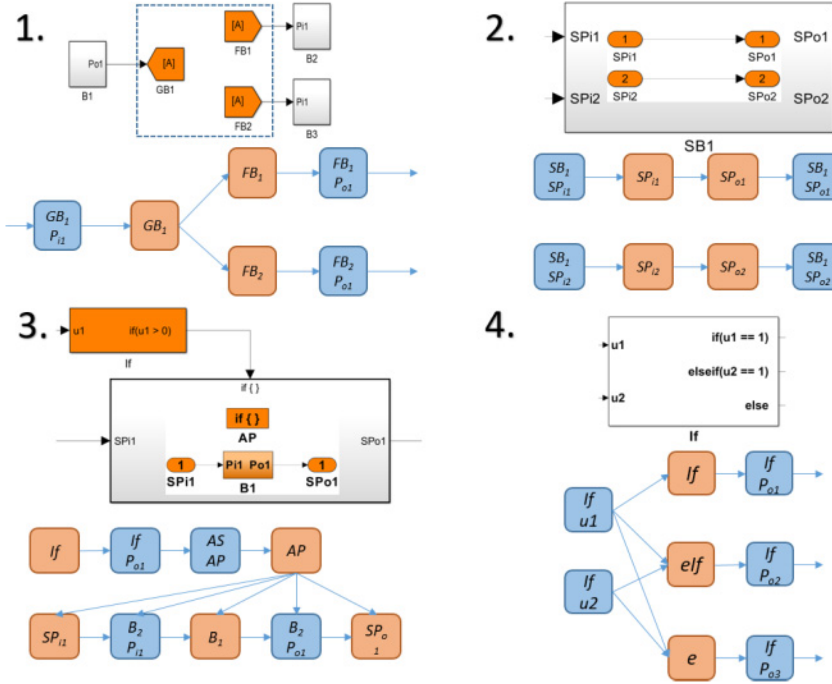

Fig. 4. Graph constructions for (1) Goto/From, (2) Subsystem, (3) Conditional Subsystems and (4) If blocks.

Subsystems. *Subsystem* blocks allow submodels to be embedded insider larger models while hiding the complexity of the former. Data is transferred into and out of the subsystem block via its submodel's Inports and Outports. Example 2 in Figure 4 depicts our treatment of Subsystem blocks. (The internal blocks of *Subsystem* SB_1 are overlaid in the figure for illustration purposes.) In the graph-construction process the Subsystem block in Simulink is omitted from the graph, but submodel is retained. Edges are introduced between the inputs and of the subsystem and their corresponding Inports and Outports from the submodel.

Conditional Subsystems. Some subsystem blocks (e.g. *Action Subsystem*, *Triggered Subsystem*) have a control-flow component to their behavior in addition to the dataflow component present in subsystems described above. Specifically, these subsystems contain *action / trigger ports* AP/TP; the subsystem is then only executed if the right trigger signal is supplied during run-time. (Trigger signals can be evaluations of if/else statements or rising/falling edges of signals, for example.) As an illustration of our treatment of these types of subsystems, consider the *If-Action Subsystem* in Example 3 in Figure 4. The If block supplies an action signal to the action port (here labeled `if{}`), which activates execution of the subsystem during the simulation step if the supplied value is "true". Our construction first treats a conditional subsystem like a regular subsystem with regard to inputs and outputs. It also creates a vertex for the action port, adds an incoming port edge to this vertex as indicated in the model, and adds an edge from the action-port vertex to *every* vertex corresponding to a block in the submodel. This is necessary since a subsystem could have e.g. Goto/From blocks that are affected by the trigger port but that do are not affected by the inputs and outputs of the subsystem.

If Blocks. *If* blocks can have multiple inputs and outputs. One output is labeled *if*; there are optionally several *else if* outputs and one *else* output. Each of these outputs has a corresponding

logical expression and is connected to a conditional subsystem block. The If-block in the Example 4 in Figure 4 has two inports, u1 and u2, and one if, one else-if, and one else output signal. During execution the if-statement $u1 == 1$ is evaluated; if it is true the corresponding subsystem is executed. If it evaluates to false the else-if statement $u2 == 1$ is evaluated and its corresponding block is executed if it is true; otherwise the else statement's block is executed. In our graph construction, each logical statement is treated as a separate vertex (if, eif, e) that is connected to the corresponding subsystem. The logical expressions are then analyzed, starting at the if-statement. Each input that occurs in the logical expression of the if-statement is connected to the corresponding if-vertex. In the example the expression of the if-statement is $u1 == 1$; therefore an edge from the $u1$ port to the if-vertex is introduced. The else-if statement is $u2 == 1$. However, in order to reach this part of the execution, the if-statement has to be false. The statement could therefore be rewritten to $!(u1 == 1) \wedge (u2 == 1)$, showing that both u1 and u2 influence the statement; edges are thus introduced from the $u1$ and $u2$ inports to the eif-vertex. The else-statement does not introduce a new logical expression, but again in order to reach it both the if and else-if statements have to be false, which requires both inputs.

4.2 Association Rule Mining

We now explain our ideas for using the information stored in the dependency graph to improve the association-rule-mining component of the invariant-generation process. The basic idea is to ensure that for any association rule of form *LHS* ⇒ *RHS* that is computed, there is a dependency between each proposition in LHS and RHS. If you recall Section 2.3, the FP-Growth algorithm first identifies the frequent item sets in the test data and then generates the association rules from these item sets. It should be noted that a large proportion of the item sets that are generated cannot produce valid association rules for our purposes. For example, consider the model in Figure 2; based on our dependency information we know that in2 has no effect on out1, and therefore any association rule of form *in1* ⇒ *out1* is spurious. Thus, an item set containing only {in2, out1} cannot be used to create a non-spurious invariant. The goal is then to make sure that these invalid item sets are not created in the first place, which also reduces the search space that the association-rule miner has to explore. We propose two different approaches to achieve this goal: *constraint-based mining* and *partitioned mining*. These are presented in more detail below.

4.2.1 Constraint-Based Mining. As a means to reduce the search space the FP-Growth algorithm permits users to define constraints on the item sets that are generated [21]. The constraints are generally either *monotone* or *anti-monotone* [19].

- A constraint is *monotone* if whenever set S violates the constraint, so do all subsets of S.
- A constraint is *anti-monotone* if whenever set S violates the constraint so do all supersets of S.

In what follows we show how to use the static-analysis information to create anti-monotone constraints that will remove all frequent item sets that result in association rules leading to spurious invariants. Since the FP-Growth algorithm creates item sets starting at size 1 up to the size of the longest transaction, the benefits of an anti-monotone constraint become clear. If one can rule out a set S one can immediately rule out any superset of S as a candidate set.

A valid association rule for our purposes has the form LHS ⇒ RHS, where the LHS mentions inputs and internal variables and the RHS consists of a single output *that is dependent on every element in the LHS.* (We could allow more than one element in the RHS, but this reduces performance without enhancing the expressive power.) The following three constraints on item sets ensure adherence to a format that results only in valid association rules.

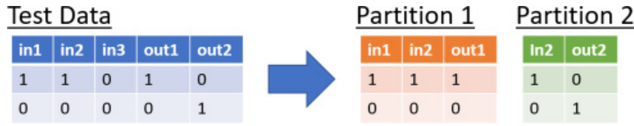

Fig. 5. Partitioning of the input data for the FP-Growth algorithm based on static analysis.

(1) If an item set has more than one output variable (RHS item) in it, the set should not be saved or used to create larger item sets.
(2) If an item set contains one RHS item and one LHS term that the RHS item is not dependent on, the set should not be saved or used to create larger item sets.
(3) If an item set does not contain an output (RHS item) yet, and there exists no output that is dependent on all the other (LHS) items in the set, the set should not be saved or used to create larger item sets.

It is easy to check that all three of these constraints are anti-monotone; any item set satisfying any of these constraints will also have all its supersets satisfying the same constraint.

Rule 3 leads to the case where some item sets do not contain a RHS but are still saved. The reason for this is that it is necessary to have all subsets of a valid item set available in order to later calculate the confidence of the association rules. These incomplete rules are marked so that they are not considered for rule creation and will only be used for the confidence calculations.

Our constraint-based mining approach involves adding the three constraints just mentioned into the FP-Growth algorithm.

4.2.2 Partitioned Mining. In this approach, instead of modifying the mining algorithm as in the constraint-based mining approach, the input data is split into sets from which only non-spurious invariants can be generated. More specifically, for each output a subset of the data is extracted that contains the output and all inputs and internal variables that our static analysis indicates the output depends on. The FP-Growth algorithm is then applied separately on the subsets, and the resulting association rules of each partition are then aggregated.

Figure 5 shows how a sample test case might be partitioned. The test case has three inputs (in1, in2 and in3) (out1 and out2). Let us assume that static analysis has shown that in1 and in2 have has a connection to out1; in2 also has a connection to out2; and in3 is not connected to any of the outputs. The approach will therefore create a partition [in1, in2, out1] and a second partition [in2, out2]. To save memory the full test case data is stored once and the partitions only store the column indices of their associated inputs, internal variables, and outputs.

Partitioning the data beforehand leads to smaller data mining problems and, in principle, should therefore speed up the data mining process and reduce the memory consumption. This speed-up could potentially be offset by the fact that the algorithm must be invoked once for each output. The experimental section of the paper investigates this trade-off.

5 EXPERIMENTAL EVALUATION

This section describes an experimental evaluation of our static-analysis-based approaches for improving invariant generation.

5.1 Hypotheses

The goal of the evaluation is to evaluate how well constraint- and partitioning-based data mining perform *vis à vis* the original approach outlined in Section 2.2, as well as with respect to each other. The following hypotheses guide our experiments.

Table 1. Un/Controlled and Dependent Variables of the Experiment

Type	Variables
Independent Controlled	Test Generator Settings, System Models, Approach (with/without improvements), Minimum Support Threshold
Independent Uncontrolled	Load on machine, Machine State, Free Memory, Random aspects of test generator
Dependent	Runtime in seconds, Memory Consumption, Jaccard Score

- H1: Using dependency information produces higher accuracy invariants and fewer spurious invariants than not using it.
- H2: Using dependency information yields reduces the search space of the data mining process and improves performance in terms of execution time and memory consumption
- H3: The constraint-based approach and partition-based approach are indistinguishable in terms of accuracy and lack of spurious invariants, and also in terms of performance.

H1 is concerned with the accuracy of the resulting invariants and H2 with the performance of the static-analysis-based approaches in this paper in comparison with the original approach, which does not use this information. H3 is used to evaluate the two static-analysis-based approaches against each other.

Accuracy is mentioned in Hypotheses 1 and (indirectly) 3. Our intention here is to compare the automatically generated invariants against a collection of invariants, which we call the *golden invariants*, for each model in our study. These invariants were produced in a three-step approach. (1) the unimproved approach was applied to generate an initial set of invariants; (2) spurious invariants were then pruned using the static analysis information and (3) the resulting invariants were manually inspected to add missing invariants that should have been added and removing invariants that are invalid. The notion of *golden invariant* is somewhat subjective as a result, and the importance of the accuracy figures is thus relative (how well do the different techniques compare to each other?) rather than absolute.

For H2, performance is measured in terms of the total execution time of the overall process as well as the execution time of data mining, and in terms of the memory consumption during data ming. For the comparison of the improvements in H3 we hypothesize that the accuracy and lack of spurious invariants should be the same. For the performance we expect differences based on the characteristics of the model: we believe that models with fewer outputs should perform better in the partitioned approach, while models with many outputs should perform better in the constraint-based approach.

5.2 Variables and Data Collection

The dependent and independent (controlled and uncontrolled) variables for the experiments are listed in Table 1. The independent controlled variables include the test-generation settings of Reactis, which allow the user some control over how Reactis creates tests. For the purposes of this study we used two different settings. Generally speaking, Reactis combines random test generation followed by a targeted phase that supplements the randomly generated tests with new tests

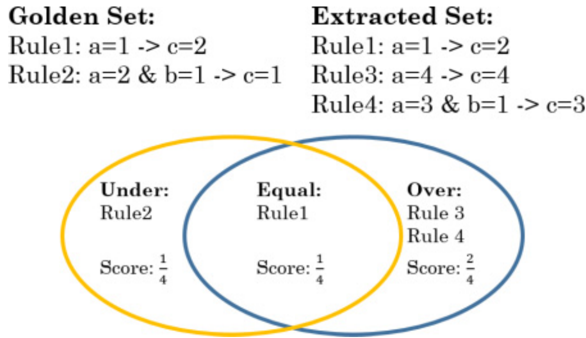

Fig. 6. Illustration of the Jaccard- and the over/under score.

that improve coverage. In our first setting, we directed Reactis to create 100 random tests with 100 steps each, and to use up to 20,000 execution steps in the targeted phase. The second test setting creates 1,000 random tests with 1,000 steps each, while again using 20,000 targeted steps. We will call the first test setting *short tests* and the second setting *long tests*. In both cases we selected all coverage criteria supported by Reactis; this means Reactis will attempt to achieve 100% coverage of all of these. Other controlled variables are the system models used in the evaluation and the settings of the requirement extraction process (e.g. turn the improvements on or off), as well as the minimum support threshold used in the data mining algorithms. The support threshold was set to 1 to guarantee that every invariant in the dataset is mined.

The dependent variables of the experiments are the runtime for each iteration in seconds (recorded for different parts of the process), the memory usage of the data-mining algorithm and the accuracy of the resulting invariants. For the execution time we record the overall time it takes to run the iteration as well as the time for the test generation, for the data mining and the time for the other processes (e.g. instrumentation). The time is captured by using a probes in the code. The peak memory usage of the data mining algorithms is measured using the MemoryLogger class of the SPMF data mining library. Spurious invariants are a subset of false positives and impact the accuracy of the resulting invariants. The accuracy is measured by comparing the resulting invariants to a set of semi-automatically created golden invariants of the systems using the Jaccard similarity coefficient [12]. The Jaccard coefficient is a set-similarity statistic that measures the overlap of two sets (the golden set of invariants, with the set produced by the framework), with a score of 0 (lowest) which signifies there is no overlap to a score of 1 (highest) which signifies complete overlap and therefore set equivalence. Two invariants are equal if both their LHS and RHS contain the same elements, any deviation (e.g. a=4 instead of a=1 on the lhs) means they are not equal and reduce the Jaccard score. However, a score below 1 can be caused by missing true invariants and/or false positives. We therefore added two more measurements based on the Jaccard coefficient. The under score indicates how much of the deviation from 1 is caused by missing invariants and the over score indicates how much is caused by false positives. This is illustrated in Figure 6; the golden set in this example contains two invariants, while the extracted set contains three. The intersection of the two sets contains only one invariant and the union of the set contains four invariants, and thus the Jaccard score of this example is 1/4. The under score is calculated by dividing the number of invariants in the complement $golden/extracted$ with the elements in the union, which in the example is also 1/4. The over score is calculated accordingly using the complement $extracted/golden$ and yields $2/4 = 1/2$.

Table 2. Model Metrics, Variables are Listed as Inputs/Internal/Outputs

Name	**Variables**	**Blocks**	**Invariants**	**Connectivity**
Cruise Control	7/2/2	83	34	9/8/8.5
Daytime Driving Light	14/0/3	52	31	9/8/8.3
Emergency Blinking	2/1/1	165	46	3/3/3.0
Exterior Light	31/4/22	515	406	17/6/11.5
Fog Light	10/3/4	59	70	10/7/8.8
High Beam Light	9/0/2	52	24	7/7/7.0
Low Beam Light	9/0/5	40	48	7/7/7.0
Parking Light	7/0/7	83	38	7/5/5.6
Position Light	9/0/7	48	30	5/5/5.0
Rear Fog Light	11/0/5	56	28	9/2/6.8
Defibrillator 1	4/3/3	184	63	7/7/7.0
Defibrillator 2	7/16/5	627	69	22/21/21.2

The independent uncontrolled variables are related to nondeterministic factors of the experimentation system, such as the load on the machine during the experiments, the available free memory, and the machine state. Another uncontrolled factor is the random aspect of the Reactis test generator, which is present in both the initial and targeted phases of the test-generation procedure. To control for these last aspects we constrain the random seeds used in the Reactis randomization process. Specifically, each experiment starts with the same fixed random seed and in each iteration the random seed is increased by one. This allows variations of test cases between the different iterations; however the random aspects of corresponding iterations in different experiments will be the same. The targeted test generation aspect depends on the instrumented invariants and cannot be fully controlled. To mitigate the impact of these uncontrolled variables, each experimental run is repeated ten times and the resulting dependent variables are then analyzed using their averaged values.

5.3 Test Applications

We use 12 Simulink models for our experiments. Ten of the models are from the automotive domain and represent parts of the lighting- and cruise-control systems of a car. The other two models are from the medical-device domain and are models of cardiac defibrillators [14]. The models vary in their complexity, with the smallest containing 40 Simulink blocks and the largest 627 blocks. The combined number of in/outputs and internal variables of the models varies from four to 57. From the 12 models five have internal state variables, while the other 7 are stateless. The number of golden invariants for the different models varies from 24 to 406. The last column shows the connectivity of the model, which is the maximum, minimum and average number of inputs and internal variables that the outputs of the system depend on. For example the Cruise Control model has two outputs; the first output is traceable via static analysis to all 9 inputs and internal variables, whereas the second one is only traceable to 8 of them, making its average number of connections from inputs to outputs 8.5.

6 RESULTS AND DISCUSSION

The experimental data can be seen in Tables 3–6. The approach was executed with the three different data-mining strategies on each of the 12 models. For evaluation of the performance values

Table 3. Jaccard Score and # of Iterations to End the Experiments

	Short Tests: Jaccard (Iterations)			Long Tests: Jaccard (Iterations)		
Model Name	**UA**	**CBA**	**PBA**	**UA**	**CBA**	**PBA**
Cruise Control	0.72 ↑0.04 ↓0.34 15	0.93 ↑0.02 ↓0.05 12	0.92 ↑0.04 ↓0.05 13	0.94 ↑0.03 ↓0.03 (4)	**0.97 ↓0.03 (4)**	**0.97 ↓0.03 (4)**
Daytime Driving Light	T/O	1.00 (8)	1.00 (8)	T/O	**1.00 (4)**	**1.00 (4)**
Emergency Blinking	1.00 (4)	1.00 (4)	1.00 (4)	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**
Exterior Light	Crash	1.00 (7.3)	1.00 (7.1)	Crash	Crash	**1.00 (4)**
Fog Light	0.17 ↑0.83 (15)	0.97 ↑0.03 (5)	0.97 ↑0.03 (5)	1.00 (11)	**1.00 (4)**	**1.00 (4)**
High Beam Light	0.29 ↑0.71 (15)	1.00 (6)	1.00 (6)	1.00 (7)	**1.00 (4)**	**1.00 (4)**
Low Beam Light	0.62 ↑0.38 (12)	1.00 (6)	1.00 (6)	0.73 ↑0.27 (5)	**1.00 (4)**	**1.00 (4)**
Parking Light	1.00 (8)	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**
Position Light	0.42 ↑0.58 (12)	**1.00 (4)**	**1.00 (4)**	0.45 ↑0.55 (5)	**1.00 (4)**	**1.00 (4)**
Rear Fog Light	T/O	1.00/8	1.00 (8)	T/O	**1.00 (4)**	**1.00 (4)**
Defibrillator 1	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**
Defibrillator 2	Crash	Crash	**1.00 (5)**	Crash	Crash	Crash

Table 4. Data Mining Time per Iteration in Seconds

	Mining Time (Short Tests)					Mining Time (Long Tests)				
Model Name	**UA**	**CBA**		**PBA**		**UA**	**CBA**		**PBA**	
Cruise Control	**0.64**	0.78	122%	0.87	136%	**8.17**	8.62	106%	20.51	251%
Daytime Driving Light	T/O	1.19	N/A	**0.71**	N/A	T/O	**28.56**	N/A	47.15	N/A
Emergency Blinking	0.052	**0.035**	67%	0.068	131%	**2.49**	2.66	107%	4.37	176%
Exterior Light	Crash	560	N/A	**22.74**	N/A	Crash	21364	N/A	**560**	N/A
Fog Light	16.45	1.46	9%	**0.98**	6%	60.15	**17.12**	28%	34.35	57%
High Beam Light	0.84	**0.24**	29%	0.39	46%	18.84	**10.92**	58%	21.22	113%
Low Beam Light	0.33	0.2	61%	**0.87**	138%	15.9	**11.7**	74%	28.06	176%
Parking Light	0.33	**0.2**	61%	0.57	173%	12.45	**11.83**	95%	38.1	306%
Position Light	0.7	**0.18**	26%	0.45	64%	17.95	**14.59**	81%	32.75	182%
Rear Fog Light	T/O	9.56	N/A	**8.47**	N/A	Crash	**77.49**	N/A	153	N/A
Defibrillator 1	**0.12**	0.17	142%	0.45	375%	8.42	**8.41**	100%	26.56	315%
Defibrillator 2	Crash	Crash	N/A	296	N/A	Crash	Crash	N/A	Crash	N/A

we followed the example of Fontana [9] and calculated paired data by comparing the results of the unimproved approach (UA) against the constraint-based (CBA) and partitioning-based approach (PBA). The result is a percentage value that indicates the increase or decrease of time or memory consumption for each of the models.

The Jaccard score is of the resulting set of invariants after the iterative approach terminated. Because some experiments of the UA would not terminate properly, due to the spurious invariants, we introduced a 15-iteration limit. Execution times are given in seconds. Memory consumption is recorded in megabyte (MB). Experiments that crashed the experimental setup are marked with *crash*. A *T/O* marks an experiment that exceeded a limit (8 hours) that we set in place to cap the execution of an iteration. We omitted separate mention of the static-analysis times, since it has to be performed only once at the beginning of the process and the impact on the performance is

Table 5. Average Time per Iteration in Seconds

	Iteration Time (Short Tests)					**Iteration Time (Long Tests)**				
Model Name	**UA**	**CBA**		**PBA**		**UA**	**CBA**		**PBA**	
Cruise Control	95.48	**79.02**	83%	79.15	83%	**735**	737	100%	746	101%
Daytime Driving Light	T/O	61.40	N/A	**61.15**	N/A	T/O	**457**	N/A	458	N/A
Emergency Blinking	52.91	52.63	99%	**52.39**	99%	574	*573*	100%	575	100%
Exterior Light	Crash	874	N/A	**338**	N/A	Crash	26616	N/A	**4487**	N/A
Fog Light	316	66.70	21%	**66.67**	21%	1719	**813**	47%	836	49%
High Beam Light	138	58.18	42%	**57.95**	42%	824	**577**	70%	581	71%
Low Beam Light	102	**57.43**	56%	58.27	57%	709	**557**	81%	594	84
Parking Light	59.06	56.17	95%	**55.81**	94%	548	**546**	100%	571	104%
Position Light	64.11	49.57	77%	**49.37**	77%	438	**407**	93%	425	97%
Rear Fog Light	Crash	76.88	N/A	**76.76**	N/A	Crash	**824**	N/A	842	N/A
Defibrillator 1	68.86	**68.41**	99%	69.13	100%	**709**	710	100%	728	103%
Defibrillator 2	Crash	Crash	N/A	447	N/A	Crash	Crash	N/A	T/O	N/A

Table 6. Peak Memory Consumption During Data Mining

	Max Memory (Short Tests)					**Max Memory (Long Tests)**				
Model Name	**UA**	**CBA**		**PBA**		**UA**	**CBA**		**PBA**	
Cruise Control	550	520	95%	**324**	59%	7168	6421	90%	**5802**	81%
Daytime Driving Light	T/O	414	N/A	**200**	N/A	T/O	5759	N/A	**5543**	N/A
Emergency Blinking	**76**	335	441%	145	191%	**2189**	2468	113%	2313	106%
Exterior Light	Crash	5195	N/A	**2265**	N/A	Crash	Crash	N/A	**12862**	N/A
Fog Light	5051	316	6%	**194**	4%	11290	**6634**	59%	7009	62%
High Beam Light	416	177	43%	**132**	32%	6834	6480	95%	**6162**	90%
Low Beam Light	525	203	39%	**144**	27%	6422	6172	96%	**5950**	93%
Parking Light	260	163	63%	**106**	41%	Crash	**6170**	N/A	7694	N/A
Position Light	590	150	25%	**115**	19%	7127	**5568**	78%	5868	82%
Rear Fog Light	T/O	**256**	N/A	270	N/A	T/O	**6576**	N/A	6995	N/A
Defibrilator 1	241	267	111%	**228**	95%	4950	5932	120%	**4443**	90%
Defibrilator 2	Crash	T/O	N/A	**20953**	N/A	Crash	T/O	N/A	T/O	N/A

minuscule compared to the testing and data mining. For example, the longest execution time of the dependency analysis was 73 milliseconds (for the Exterior Light model).

6.1 Hypothesis 1: Accuracy

For H1 we are evaluating whether or not the accuracy was improved with the introduction of CBA and PBA. Table 3 shows the Jaccard score after the last iteration of each experiment and how many iterations it took to finish. As can be seen, the UA crashed for two of the 12 models due to lack of memory (which means it needed more than the 32GB available to it) and timed out for two

more. For the 10 models that it was able to process fully, in the short tests Reactis setting it has a lower accuracy in eight models and an equal accuracy in two. For the long tests the UA has a lower accuracy in six models and equal accuracy in four. This indicates that H1 is true, because the accuracy of the improvements is always higher or at least as high as the UA. The unimproved version often also requires more iterations than the improved versions, since the improvements can eliminate spurious invariants early on, meaning the test generator can focus on a smaller number of invariants.

The UA has issues with spurious invariants in five models for the short tests; however for two of these models in the long tests the accuracy is the same as the improved version. This suggests that the issue with spurious invariants can be overcome, at least partly, with additional random test data, but it is not guaranteed. The results also seem to provide further evidence to the observation in [1] that better (in this case more tests) improve the accuracy.

6.2 Hypothesis 2: Performance

For H2 we are evaluating whether or not the performance in terms of execution time and maximum memory consumption was improved with the introduction of CBA and PBA. Table 4 shows the average execution times for the data-mining process in each iteration of an experiment; the average time for the full iteration (test generation + data mining + instrumentation) can be seen in Table 5 and the maximum memory consumption during data mining in Table 6. For the short tests the UA crashes for two out of 12 models; it times out for two more. It is slower in six and faster in two models. For the models where the UA does not crash CBA is 38% faster and PBA is 34% slower on average. For the long tests the UA also crashes and times out two times, respectively. It is slower in six and faster in two models. The CBA miner is 19% faster on average, and PBA 97% slower. The cases where the UA is competitive with CBA are mostly the small- to medium-sized models, with highly connected in/outputs. This is to be expected since the improvements are targeted at removing overhead from unconnected values. The slightly faster performance in some of the cases with high connectedness indicates that there is some overhead cost associated with the improvements. This was expected, especially for PBA, since it has to be executed once for every output. Also, checking the constraints for each item set in CBA incurs a cost that only pays off if it reduces the search space.

The overall time per iteration is the time spent on test generation, data mining and other tasks in the framework (e.g. instrumentation). For small- to medium-sized models with high connectivity the execution time of the data miner and the overall iteration time is similar (e.g. Parking Light). However, for models where the UA infers many spurious invariants (indicated by the low accuracy of the results, e.g. High Beam Light) the execution time of the iteration can be slowed down considerably. This is due to the fact that test generator has to spend time trying to disprove the spurious invariants. The *Daytime Driving Light* and *Rear Fog Light* are extreme cases for this behavior. They produce over 3000 spurious invariants, and the test generator therefore breaks the eight-hour limit that we set for each iteration.

The UA uses more memory in seven cases and less in one case for the short and long tests. On average CBA uses 3% more memory for the short and 7% less for the long tests. However, the CBA result is skewed by the one outlier in the *Emergency Blinking* model, which has a very low memory footprint. Without this model CBA is on average 46% faster for the short and 9% for the long test. PBA uses 59% less memory in the short test and 18% in the long tests. The crashes of the system are caused by lack of memory. These results show that H2 is not always true. For some cases where the system is highly connected there can be performance penalties due to the overhead associated with checking the constraints and a large overhead for the partitioning of the data that can make the improvements slightly slower. However, for the small- to medium-sized

models the data-mining time is only a small factor of the overall time and even though there are improvements, they are negligible in comparison to the overall iteration time. But the improvements can have a significant impact for larger models both in terms of execution time and memory consumption. The unimproved approach was not able to handle the data mining for the two largest models. The two magnitudes of additional test data from the short to the long tests have a large impact on the execution time and the memory consumption of the approach. The FP-Growth algorithm only has to read the data twice and is therefore not bounded so much by the amount of data than other association-rule-mining algorithms. However, generating and parsing the tests can take considerable amounts of time for the larger tests. Also all the test data is aggregated in each iteration and is kept in memory which can also take up 2-7GB of memory (depending on the number of variables). One solution to this problem could be the addition of mining approaches that allow to add data incrementally; this way only the new test data would have to be read and stored in memory and could be discarded after the iteration.

6.3 Hypothesis 3: Improvement Comparison

For H3 we are evaluating whether or not CBA and PBA are performing the same in terms of accuracy and execution time/memory consumption. The results in Table 3 show that the accuracy of the two approaches is the same except for one case. In the short tests of the cruise-control model CBA has an accuracy of 0.95 and takes 12 iterations and PBA has an accuracy of 0.94 and takes 13 iterations. We analyzed the data and applied the test data from PBA to CBA but they produce the same results. However, what is different is the order in which the invariants are recorded. The order of the invariants could influence the generated tests from Reactis and explain this small discrepancy.

For the small- and medium-sized models (Table 4) CBA is usually better since PBA has to executed once for each output. This is especially apparent for the long tests. For the larger models, however, PBA performs much better. PBA is the only algorithm that can handle the second defibrillator model and for the Exterior Light model PBA mines the invariants in 23s for the short and 560s for the long tests. It takes CBA 560s and 21,070s respectively to mine the same data. Furthermore, PBA tends to use less memory than CBA, although this is not true for all models. Thus, the first part of H3 is true; CBA and PBA have the same accuracy. The performance of the two approaches depends heavily on the model characteristics and the amount of test data; also PBA tends do be less memory-intensive for systems that have less connectivity and therefore the second part of H3 is not true. Analysis of the connectedness could be used to determine dynamically which of the algorithms to use for a model. Except for fully connected models the static analysis information always proved useful. Adapting the idea behind CBA requires that the algorithm supports anti-monotone constraints and requires changes in their source code. The idea behind PBA on the other hand can be easily adapted to other data-mining algorithms. For future work we adapted it to a decision-tree-based algorithm, without having to make any changes to the algorithm itself.

7 THREATS TO VALIDITY

The discussion about potential threats to validity of the experimental evaluation presented in this paper is based on Wohlin et al's [23] four-class layout that includes: threats to internal, external, construction, and conclusion validity.

7.1 Threats to Internal Validity

We considered two potential problems for internal validity. (1) Do the different implementations of the data miner (unimproved, constraint-based and partition-based) work as intended? (2) Are there any inefficiencies in the implementation that could explain the results? In order to evaluate

if all the algorithms do the right thing we tested them with data for which we know the resulting invariants. To avoid accidental inefficiencies we used the same implementation of the FP-Growth algorithm for all three versions. The difference between them is a boolean flag that activates the additional constraint checks but otherwise they share the same code base. We also profiled the code with JProfiler[5] to identify potential bottle necks.

7.2 Threats to External Validity

The specific implementation of our approach in this paper analyzes Simulink models, and we applied it on models from the automotive and medical-device domain. However, it should be applicable to other Simulink models as well. In fact, we believe that the approach is generally applicable as long as the requirements in Section 2.2 are met.

7.3 Threats to Construct Validity

Using incorrect measures in the experiments is a threat to construct validity. For the performance of the improvements we measured the time and the memory consumption, which are direct measures and therefore good indicators for relative performance. For measuring accuracy we use the Jaccard set similarity score to compare the generated set of invariants against a golden set we provided and validated as described in Section 5.1. To make sure that the measures were not influenced by any uncontrolled factors (like the state of the experimentation machine) we repeated all experiments 10 times and averaged the results.

7.4 Threats to Conclusion Validity

In order to mitigate self bias, we applied the approach to a variety of models where some of the models played to the strength of the approaches (inputs and internal variables that are not connected to some of the outputs) and some represented the worst-case scenario (all inputs, internal variables are connected to all the outputs).

8 RELATED WORK

Program invariants have been used to reduce the search space of model checkers [6], for the analysis of log files [5], or for specification extraction [8]. This work improves the accuracy and scalability of a specification-extraction approach [1] by incorporating information from static data- and control-flow analysis into the invariant-mining algorithm. Other specification-extraction approaches, such as the Refinement Calculus of Reactive Systems (RCRS) [7], creates contracts of a Simulink model in a bottom-up modular fashion, producing a specification that describes the global behavior of the model. Rather than specifying globally what all the behaviors are, our approach creates several invariants that are partial descriptions of this global behavior that can be understood independently.

Translating Simulink models into other modeling notations and languages has often been done for the purpose of formal verification [2, 18]. Pantelic et al. [20] present a tool suite that can extract interfaces from Simulink submodels using static data/control-flow analysis. Instead of extracting interfaces our approach extracts invariants of the whole system. The Artshop tool [11] is a suite of tools for analysis, reporting and inspection of Simulink models. It has the capability to do static dependency/control-flow analysis, which is used to create slices of models. However, it does not have a specification extraction tool. Lublinerman et al. [15–17] introduce several methods for dependency analysis of Simulink models that they use for modular code generation. Our graph representation and transformation process is similar to their *block-based dependency analysis*, but

[5] https://www.ej-technologies.com/products/jprofiler/overview.html

we are not employing it to give semantics to a diagram and are only interested in computing the dependencies between elements in a model. Their work also presents a *input-output dependency analysis* based on bipartite graphs. However, this is not applicable in our context, since we also need information about internal variables of a system.

The GoldMine tool [22] extracts temporal and propositional invariants for hardware designs using a decision-tree algorithm. It uses static analysis of the hardware design and formal verification to remove spurious invariants produced by the data miner. In our approach we created a static analyzer for Simulink models that can identify data and control dependencies between inputs, internal variables and the outputs of the system and the assertions are used to infer specifications of the system.

9 CONCLUSION

This paper shows how data- and control flow dependencies in Simulink models may be computed and used to improve the mining of invariants from models. The data- and control- flow information is obtained by transforming a Simulink model into a directed graph using rules presented in this work. The resulting graph can then be searched to identify the dependencies between the inputs, internal variables and outputs of a system. This information is then used to introduce two improved versions of the data-mining process of an existing invariant-extraction technique. The improvements eliminate spurious invariants and reduce the search space of the data miner, thereby increasing the accuracy and decreasing execution time and memory consumption.

We also report the results of an experimental evaluation on 12 Simulink models from the automotive and medical-device domains. The experiments show that the improvements made it possible to analyze all models that caused the original approach to crash due to memory overflows. For the other models the new techniques reduced the execution time and the memory consumption (by %39/%42). Furthermore, the improvements increased the accuracy of the approach by removing the false positives that were inferred by the data miner. One technique (the constraint-based data miner) was fastest for medium-sized models and more test data, whereas the other technique (the partition-based data miner) had lower memory consumption and scaled better for larger models.

For future work we are planning to integrate algorithms that can handle numerical variables without manual abstractions and integrate the static analysis information into these algorithms. We also intend to compare mined specifications against actual system specifications. For this comparison we are developing an inspection tool that helps the user to better analyze the mined specifications.

ACKNOWLEDGMENTS

The authors wish to thank Rahul Mangharam, Zhihao Jiang and Mikael Lindvall for supplying us with Simulink models of real world systems and to Philippe Fournier-Viger for the SPMF data mining library and the great support on his website.

REFERENCES

[1] Chris Ackermann, Rance Cleaveland, Samuel Huang, Arnab Ray, Charles Shelton, and Elizabeth Latronico. 2010. Automatic requirement extraction from test cases. In *Proceedings of the First International Conference on Runtime Verification (RV'10)*. Springer-Verlag, Berlin, Heidelberg, 1–15. http://dl.acm.org/citation.cfm?id=1939399.1939401

[2] Aditya Agrawal, Gyula Simon, and Gabor Karsai. 2004. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science* 109 (2004), 43–56. DOI: http://dx.doi.org/10.1016/j.entcs.2004.02.055 Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[3] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. *SIGMOD Rec.* 22, 2 (June 1993), 207–216. DOI: http://dx.doi.org/10.1145/170036.170072

[4] Fabrizio Angiulli, Giovambattista Ianni, and Luigi Palopoli. 2001. On the complexity of mining association rules. In *SEBD*. 177–184.

[5] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. 2011. Mining temporal invariants from partially ordered logs. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML'11)*. ACM, New York, NY, USA, Article 3, 10 pages. DOI: http://dx.doi.org/10.1145/2038633.2038636

[6] Xueqi Cheng and Michael S. Hsiao. 2008. Simulation-directed invariant mining for software verification. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*. ACM, New York, NY, USA, 682–687. DOI: http://dx.doi.org/10.1145/1403375.1403541

[7] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. 2016. *Compositional Semantics and Analysis of Hierarchical Block Diagrams*. Springer International Publishing, Cham, 38–56. DOI: http://dx.doi.org/10.1007/978-3-319-32582-8_3

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1–3 (Dec. 2007), 35–45. DOI: http://dx.doi.org/10.1016/j.scico.2007.01.015

[9] Peter Fontana. 2015. *Towards a Unified Theory of Timed Automata*. Ph.D. Dissertation. University of Maryland.

[10] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhihong Deng, and Hoang Thanh Lam. 2016. The SPMF open-source data mining library version 2. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 36–40.

[11] Thomas Gerlitz, Norman Hansen, Christian Dernehl, and Stefan Kowalewski. artshop: A continuous integration and quality assessment framework for model-based software artifacts. In *Tagungsband des Dagstuhl-Workshops*. 13.

[12] Lieve Hamers, Yves Hemeryck, Guido Herweyers, Marc Janssen, Hans Keters, Ronald Rousseau, and André Vanhoutte. 1989. Similarity measures in scientometric research: The jaccard index versus salton's cosine formula. *Information Processing & Management* 25, 3 (1989), 315–318.

[13] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *SIGMOD Rec.* 29, 2 (May 2000), 1–12. DOI: http://dx.doi.org/10.1145/335191.335372

[14] Z. Jiang, M. Pajic, A. Connolly, S. Dixit, and R. Mangharam. 2010. Real-time heart model for implantable cardiac device validation and verification. In *2010 22nd Euromicro Conference on Real-Time Systems*. 239–248. DOI: http://dx.doi.org/10.1109/ECRTS.2010.36

[15] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. 2009. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. ACM, New York, NY, USA, 78–89. DOI: http://dx.doi.org/10.1145/1480881.1480893

[16] R. Lublinerman and S. Tripakis. 2008. Modular code generation from triggered and timed block diagrams. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. 147–158. DOI: http://dx.doi.org/10.1109/RTAS.2008.12

[17] Roberto Lublinerman and Stavros Tripakis. 2008. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*. ACM, New York, NY, USA, 1504–1509. DOI: http://dx.doi.org/10.1145/1403375.1403736

[18] B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. 2006. Tool for translating simulink models into input language of a model checker. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering (ICFEM'06)*. Springer-Verlag, Berlin, Heidelberg, 606–620. DOI: http://dx.doi.org/10.1007/11901433_33

[19] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. 1998. Exploratory mining and pruning optimizations of constrained associations rules. *SIGMOD Rec.* 27, 2 (June 1998), 13–24. DOI: http://dx.doi.org/10.1145/276305.276307

[20] V. Pantelic, S. Postma, M. Lawford, A. Korobkine, B. Mackenzie, J. Ong, and M. Bender. 2015. A toolset for simulink: Improving software engineering practices in development with simulink. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 1–12.

[21] Jian Pei, Jiawei Han, and L. V. S. Lakshmanan. 2001. Mining frequent itemsets with convertible constraints. In *Proceedings 17th International Conference on Data Engineering*. 433–442. DOI: http://dx.doi.org/10.1109/ICDE.2001.914856

[22] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. GoldMine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 626–629. http://dl.acm.org/citation.cfm?id=1870926.1871074

[23] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA.

Received April 2017; revised June 2017; accepted July 2017