Generalized Observational Slicing for Tree-Represented Modelling Languages

Nicolas E. Gold
n.gold@ucl.ac.uk
Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK

David Binkley
binkley@cs.loyola.edu
Department of Computer Science
Loyola University Maryland
Baltimore MD 21210–2699, USA

Mark Harman
mark.harman@ucl.ac.uk
Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK

Syed Islam
syed.islam@uel.ac.uk
School of Architecture, Computing and Engineering (ACE)
University of East London
London E16 2RD, UK

Jens Krinke
j.krinke@ucl.ac.uk
Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK

Shin Yoo
shin.yoo@kaist.ac.kr
School of Computing
Korea Advanced Institute of Science and Technology
Daejon 34141, Republic of Korea

ABSTRACT

Model-driven software engineering raises the abstraction level making complex systems easier to understand than if written in textual code. Nevertheless, large complicated software systems can have large models, motivating the need for slicing techniques that reduce the size of a model. We present a generalization of observation-based slicing that allows the criterion to be defined using a variety of kinds of observable behavior and does not require any complex dependence analysis. We apply our implementation of generalized observational slicing for tree-structured representations to Simulink models. The resulting slice might be the subset of the original model responsible for an observed failure or simply the sub-model semantically related to a classic slicing criterion. Unlike its predecessors, the algorithm is also capable of slicing embedded Stateflow state machines. A study of nine real-world models drawn from four different application domains demonstrates the effectiveness of our approach at dramatically reducing Simulink model sizes for realistic observation scenarios: for 9 out of 20 cases, the resulting model has fewer than 25% of the original model's elements.

CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**; *Extensible Markup Language (XML)*; *Software testing and debugging*;

KEYWORDS

Slicing, ORBS, Simulink, MATLAB, Observational Slicing

ESEC/FSE'17, September 4-8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106304

ACM Reference format:
Nicolas E. Gold, David Binkley, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2017. Generalized Observational Slicing for Tree-Represented Modelling Languages. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106304

1 INTRODUCTION

Executable models are widely used in software engineering as well as other engineering domains to prototype, communicate, reason about, and simulate complex systems. Two reasons for their widespread use are that they support the representation of, and permit engagement with, larger and more complex domains than would be tractable using traditional (lower-level) programming languages. Nonetheless, models are still plagued by familiar problems caused by their complexity. For example, model comprehension can easily become a challenge as model size grows. As such, there is a need for analytical techniques that can address problems such as test case generation [27, 40, 45], fault localization [27], impact analysis [1], slicing [6, 43], and clone detection [19, 41]; techniques that mirror those developed for programs.

One particular challenge, borne of the implicit size of the systems captured by models, is a need for techniques such as program slicing (or perhaps *model slicing*) that can extract from a large model those elements that pertain to the computation (e.g., a fault) of interest. Such techniques help to uncover the causes of the inevitable errors found when modeling complex systems. Example model slicing techniques include those based on dependence analysis [43] and model projection [6].

This paper presents a new method, *Tree Observational Slicing (Tree-ORBS)*, for dynamically slicing models where the only prerequisite is that the model is represented using a tree-structured representation (e.g., XML). Tree-ORBS is inspired by ORBS [11, 12, 54], an observation-based slicing technique for traditional programming languages. It is capable of slicing away parts of a model not required to capture a specified behavior. In addition to behaviors

specified using explicit parts of the model (e.g., the values flowing from one component to another), the specified behavior can also be a *property of the output* (e.g., the presence of a particular tone from a music simulation model), or even *of the execution itself* (e.g., a warning from the run-time environment rather than the model).

In this paper, we focus on a widely-used product for modeling that uses a tree structured representation: Mathworks' Simulink [47], part of the MATLAB software suite. Simulink provides a graphical simulation environment in which complex discrete and continuous systems can be constructed and simulated under a range of experimental conditions. Simulink models have been the subject of research for various model analysis techniques including test-case generation and fault localization [27, 40, 45], clone detection [19, 41], and quality assessment metrics [38]. Although Simulink is our focus here, the algorithm is not tied to Simulink and has been subsequently successfully applied to other XML-based representations [13] (e.g., srcML [16]).

As an example application of Tree-ORBS, consider using it with a system for finding test data that triggers a fault. One such tool for Simulink, described by Holling, Pretschner, and Gemmar, is 8Cage [27, 42]. The combination of 8Cage and Tree-ORBS provides the ability to undertake automated end-to-end test-case generation and model reduction for fault localization. Given a fault of interest, the first step runs 8Cage, which produces test data that triggers the fault. Then, using this data, Tree-ORBS can produce a reduced model that includes only those components that contribute to the production of the fault.

In support of such applications, this paper makes the following contributions

- *Observational Slicing* – a generalization of observation-based slicing that supports different types of observation and criteria.
- An algorithm, Tree-ORBS, for observational slicing of tree-structured representations.
- An implementation of Tree-ORBS for XML-based languages such as Simulink (XML-ORBS).
- A case study that demonstrates the utility of Tree-ORBS by exhibiting the end-to-end fault localization scenario outlined above.
- Empirical studies that demonstrate the application, operation, and characteristics of XML-ORBS applied to a collection of Simulink models (in some cases including Stateflow).

2 BACKGROUND

2.1 Slicing Programs and Models

Program slicing is a technique for deleting parts of a software system that are irrelevant to a chosen slicing criterion [7, 44, 52]. There are many forms including static, dynamic, quasi-static, conditioned, amorphous, and syntax-preserving [44]. By focusing on the criterion, slicing has found many applications, including testing [10, 25], debugging [33, 53], maintenance [21, 23], re-engineering [14], reuse [9, 15], comprehension [18, 32, 50] and refactoring [20].

The present paper is inspired by a recently-introduced form of dynamic slicing known as Observation-Based Slicing [11, 12, 54], which dynamically slices multilingual systems using *speculative deletion* of system elements as the slicing operation. Such deletions are checked against observations with respect to the behavior as specified by the criterion. Those deletions for which there are no observable changes for the criterion are accepted, while those that have an observable effect are discarded (so the corresponding program elements are retained and the speculative deletion is rejected). Observation-based slicing has the advantage that it can capture dependencies overlooked by previous slicing techniques, which fail to account for observation (preferring some more abstract dependence model) [12]. Observation-based slicing produces inherently smaller slices than static slicing, because it is based on dynamic observation, but it can also produce smaller slices than traditional dynamic approaches because of its focus on observation (rather than dynamically-traversed statically-defined dependence) [11, 30].

One important feature of the ORBS implementation of observation-based slicing [11] is the way that focusing on deleting lines instead of statements liberates the slicing algorithm from the need for detailed (and thus expensive and brittle) semantic analysis; lines can be deleted and the effect simply observed, a property from which ORBS derives its language independence. Furthermore, using only speculative lexical deletion is language-independent, thereby ORBS is able to slice systems expressed in multiple languages. In contrast, such multilingual slicing is a considerable challenge for existing (dependence analysis based) slicing techniques. However, hitherto, observation-based slicing has only been applied to programs, not to models.

2.2 Slicing Simulink

This paper presents the first tree-based *observational* slicing technique and reports on experiments with its implementation using a collection of MATLAB/Simulink models. A Simulink model, which is saved textually as an XML file (or files if Stateflow is included), is visually represented as a diagram with functional blocks (possibly encapsulating sub-systems) connected by lines representing the flow of data during simulation. We treat the XML file(s) as the model's 'source code'. Associated with the model are a range of parameters governing the way the simulation is run and how the model interacts with external devices, files, and the underlying MATLAB workspace. Simulink models can interface with Stateflow state machines, which then share data.

There are several existing approaches to slicing Simulink models [22, 39, 43, 46], but none of them uses observational slicing, leaving open the question as to what extent the advantages reported for observation-based program slicing extend to observational model slicing. Of course, an XML file could be regarded as line-based source code to which the original ORBS implementation of traditional observation-based slicing [11] could be applied. However, this approach would be suboptimal as it would not take advantage of the tree-based nature of XML where whole sub-trees can be pruned rather than individual lines.

Without sacrificing language independence, but exploiting the tree structure of XML, we introduce a tree-based observational slicing algorithm, Tree-ORBS. We implement the algorithm for Simulink models and evaluate this on a collection of real-world Simulink models. Unlike non-observational alternatives, the resulting slices are fully executable. Thus they can be loaded into

MATLAB/Simulink and used in place of the original. Our implementation can slice Stateflow state machines as part of a Simulink model.

3 A GENERALIZED FRAMEWORK FOR OBSERVATION-BASED SLICING

The concepts of static, dynamic, and observation-based slicing work well for traditional programming languages (e.g. C, Java) where variables and statements are well defined. For other programming languages and criteria, the definitions of slicing and the corresponding implementations must be changed accordingly. Instead of giving yet another specific definition, we generalize observation-based slicing to accommodate different observations that can be made over a program. The original definition of an observation-based slice is based on comparing execution trajectories thus:

(Trajectory) Observation-Based Slice [11]: An *observation-based* slice S of a program P on a slicing criterion $C = (v, l, \mathcal{I})$ composed of variable v, line l, and set of inputs $\mathcal{I}$, is any executable program with the following properties:

(1) The execution of P for every input I in $\mathcal{I}$ halts and produces a sequence (a trajectory) of values $V(P, I, v, l)$ for variable v at line l.
(2) S can be obtained from P by deleting zero or more statements from P.
(3) The execution of S for every input I in $\mathcal{I}$ halts and produces a sequence of values $V(S, I, v, l)$ for variable v at line l.
(4) $\forall_{I \in \mathcal{I}} V(P, I, v, l) = V(S, I, v, l)$.

In this definition the execution is *observed* using the trajectory of values that a variable (or set of variables) produces for a specific input. More generally, one can envision an observer $O(P, I)$ that extracts from program P some subset of the behavior ("the behavior of interest") for a given input I. Furthermore, the observed behavior need not be exactly matched, and thus rather than equality, the relation between the behavior of the original program and its slice need only be related by a matching relation R. Using O and R, *Generalized Observational Slicing* can be defined as follows:

Generalized Observational Slice: A *generalized observational* slice S of a program P on a slicing criterion $C = (O, R, \mathcal{I})$ composed of an observer O, a matching relation R, and a set of inputs $\mathcal{I}$, is any executable program with the following properties:

(1) The execution of P for every input I in $\mathcal{I}$ halts and produces the observed behavior $O(P, I)$.
(2) S can be obtained from P by deleting zero or more elements from P.
(3) The execution of S for every input I in $\mathcal{I}$ halts and produces the observed behavior $O(S, I)$.
(4) $\forall_{I \in \mathcal{I}} O(S, I) \sim_R O(P, I)$.

In the above definition, the program P can be any executable entity and the observer O can be any observation made about P. A simple instantiation defines the observer $O(P, I)$ as the output of a program P when P is run on each input $I \in \mathcal{I}$. If the matching relation, R, is *equality*, then the corresponding generalized observational slice S is P after unused code (w.r.t. the set of inputs $\mathcal{I}$) is removed.

Dynamic slicing can also be defined as an instance of generalized observational slicing: the trajectory of values of a variable v at a location l for input I to program P is $V(P, I, v, l)$ as defined for trajectory observation-based slicing. Given the criterion $C = (v, l, \mathcal{I})$ for a trajectory observation-based slice, the observer is $O(P, I) = V(P, I, v, l)$ and the matching relation R is equality for the generalized observational slice.

As a third example, consider a test suite $\mathcal{T}$ where each test case $T \in \mathcal{T}$ includes a model input T_I and an expected output. One can define a test-focused version of generalized observational slicing in which $O(P, T_I)$ is the observation of whether or not P passes test case T. The matching relation can either be strict, where all tests must yield the same result for the slice S and the original program P, in which case the matching relation R is equality. Alternatively, the matching relation can be *relaxed* where all tests that the original program P passes must also be passed by the slice S, but any test that the original program P fails is allowed to be passed by S. Let $\mathcal{I} = \{T_I | T \in \mathcal{T}\}$ and $X(I)$ be PASS or FAIL and $O(P, I) = X(T_I)$. The strict matching relation is equality (R is '=') and the relaxed matching relation is $R = \{(\text{PASS}, \text{PASS}), (\text{FAIL}, \text{FAIL}), (\text{FAIL}, \text{PASS})\}$.

A similar generalization extended the ORBS algorithm to slice picture description languages [54] where, when the sliced program is rendered, it produces a specified pattern (the criterion). Using generalized observational slicing, matching the specific pattern is captured by an observer O that checks if the pattern exists in the rendered image using equality for the matching relation R. The actual implementation uses a template matching score capturing how well the rendered image matches the specified template (the observer O) and the matching relation is less-or-equal-than (so that the rendered slice can match the template better (but no worse) than the rendered original picture description).

Generalized observational slicing is better suited for models than the original definition of observation-based slicing because models do not necessarily have the concepts of 'variables' or 'lines' (locations). Moreover, we can instantiate the definition of generalized observational slicing for models in a similar way to traditional programs. Examples include

A traditional slice. For a given element, E of model P, observer O extracts the trajectory of values produced for an input I at E and the matching relation, R, is equality.

Program Specialization. The observer O extracts all output of model P for input I and the matching relation, R, is equality. In this case, the slice is a variant of P specialized to $\mathcal{I}$. For example, if $\mathcal{I}$ involved the computation of distances in meters then the slice would work for (is specialized to) the subset of inputs that use meters.

Fault Localization. The observer O extracts (some subset of) the warning and error output from the *model execution environment* when executing model P on input I, and the matching relation is either equality and thus the same errors must be produced, or "non-empty subsequence" in which case at least one of the errors must be produced.

Non-termination Removal. The observer O extracts (some subset of) the output (or the warning and error output) of the model execution environment for the execution of model P for input I, and the matching relation is prefix thus allowing the slice to continue executing where P has entered an infinite loop or abnormally terminated.

We will make use of the flexibility in generalized observational slicing in defining Tree-ORBS, the algorithm for tree-structured representations, and its implementation XML-ORBS, which we use to slice Simulink models under various observations.

4 MODEL SLICING

This section presents our algorithm for slicing models and then describes details of the implementation. To begin with, Figure 1 presents the Tree-ORBS algorithm, which satisfies the definition of generalized observational slicing for tree-structured representations. The algorithm adopts a similar overall approach to ORBS but differs in its deletion-target selection and traversal strategy in order to support tree structures. It takes five inputs although the final input is optional. These five include the model to be sliced M, the slicing criterion consisting of an observer O, a matching relation R, and a set of inputs $\mathcal{I}$. The observer executes the candidate slice and returns the result of the observation for a specific input. Finally, the optional input is a start node that specifies the subtree at which the algorithm should start (the default start node is the root node).

The algorithm uses several auxiliary functions: DELETE removes the subtree rooted at component c from the tree representation of model M. CHILDREN returns the children of component c. Finally, APPEND, DEQUEUE, and EMPTY are straightforward queue operations. The algorithm begins by saving the observations that result from executing model M using each input. This output forms the oracle against which subsequent executions are compared. Like ORBS, Tree-ORBS then repeatedly passes over the nodes of the tree attempting to delete nodes until no further deletions are possible. On each iteration it traverses the tree in breadth-first order (children are appended/traversed left to right in a breadth-first greedy traversal). The processing of each component c of the tree speculatively deletes the subtree rooted at c to produce a candidate slice. The subtree rooted at c is permanently deleted if the observations for all inputs match the oracle. Otherwise c's children are appended to the breadth-first worklist. If no deletions are made during an iteration, the slice is complete. Similar to traditional ORBS, the deletion order has an impact. TreeORBS produces a 1-minimal slice, however, there may be other 1-minimal slices (the full state-space is therefore not traversed).

Based on the original ORBS implementation, we implemented the Tree-ORBS algorithm to work with XML-represented models and refer to the resulting implementation as XML-ORBS. It uses a mixture of shell-scripts and Python code to set up projects for analysis and undertake the slicing. To improve efficiency, XPath axes can be supplied to identify the start node and a stop list. The default start node is the root of the tree. The stop list specifies node types that the slicer should ignore; it is empty by default. The stop list can be used to instruct XML-ORBS not to attempt to delete nodes of a specific type, which is useful, for example, to prevent positional properties from being removed. Finally, in addition to model input, each input $I \in \mathcal{I}$ includes environmental inputs and execution settings.

Applying XML-ORBS to Simulink models requires some additional configuration to enable XML-ORBS to start MATLAB and

```
TreeORBSlice(M, O, R, 𝓘, N)
Input: model M; the criterion consisting of observer O,
matching relation R, and inputs 𝓘; and a start node N
Output: A slice, S, of M for C = (O, R, 𝓘)
(1)  foreach I ∈ 𝓘
(2)      V_I ← O(M, I)
(3)  repeat
(4)      nothingDeleted ← True
(5)      q ← APPEND(empty_queue, [N])
(6)      while ¬ EMPTY(q)
(7)          c ← DEQUEUE(q)
(8)          M' ← DELETE(M, c)
(9)          deletable ← True
(10)         foreach I ∈ 𝓘
(11)             V' ← O(M', I)
(12)             if V_I ≁_R V'
(13)                 deletable ← False
(14)         if deletable
(15)             M ← M'
(16)             nothingDeleted ← False
(17)         else
(18)             q ← APPEND(q, CHILDREN(c))
(19) until nothingDeleted
(20) return M
```

Figure 1: Tree-ORBS Algorithm

Simulink, execute models, and retrieve trajectories. A typical configuration merely requires defining an observer (e.g., by an execution script that exposes values of properties of interest to Tree-ORBS). Our implementation uses MATLAB scripts to prepare the workspace, execute the model, and extract values for TreeORBS. An observer can be as simple as a statement printing a value of interest. The matching relation can be a simple equality test (e.g., trajectories must be identical), or more complex (e.g., filter error logs and check subset containment). The advantage is that the core algorithm does not need to be modified for different properties or tree-structured languages (a line-oriented domain-specific language would be more appropriately sliced by original ORBS).

An *Executer* class in the XML-ORBS script provides an API through which application-specific execution environments can be managed. MATLAB scripts are used to provide the bridge between XML-ORBS and the model (thus forming part of the model instrumentation). Owing to the comparatively long start-up time for MATLAB/Simulink with each model execution, one version of the *Executer* class actively manages a running MATLAB instance, restarting it only when a crash or a timeout occurs. Although this 'fast' executer offers significant runtime savings over the 'generic' version, some models exhibited non-deterministic behavior under this executer despite having deterministic behavior when used with the generic executer (perhaps due to some internal persistent state within Simulink).

Recall that the generalized framework requires model evaluation to terminate for all $I \in \mathcal{I}$. This requirement cannot be guaranteed even when simulation start and end times are specified because a crash may occur causing MATLAB to drop back to the interactive

shell and thus appear not to halt. To account for this possibility, a conservative timeout value is used. It is assumed that any execution taking longer than the timeout is non-terminating and thus the current model is not a viable slice.

5 CASE STUDY

As a case study, we consider the application of XML-ORBS to the problem of fault localization using the example provided by Holling, Pretschner, and Gemmar [27]. This example was used to illustrate their technique for test-case generation (the *8Cage* model described in Table 2). Their algorithm is able to construct test cases that induce particular faults at selected blocks in a Simulink model (the values for the specific test cases used are described in a video accompanying their paper [26]). They identify three faults in the model, an absolute value overflow, a division overflow, and a threshold violation. From these they generate inputs to trigger each fault. In the following, we demonstrate how our approach is used to reduce the 8Cage model to the sub-model responsible for the failures. Since this is an experiment relying on execution properties (the introduction of failures), XML-ORBS is configured to use an observer that extracts warnings produced by the execution environment. Key to this application is that XML-ORBS produces *executable* slices.

For each of the three scenarios (one per fault), 8Cage produces a set of inputs that trigger the corresponding fault. Executing the model with each of these inputs generates a list of warnings. We apply XML-ORBS with a strict matching relation so that the slice will generate the same list of observed warnings for each input. The results are shown in Table 1 under the heading "Slices for all failures with strict matching relation." *Block Count* and *XML Lines* refer to the number of elements of the diagram (model) remaining (the *Total Elements* count is the sum of these plus the number of systems in the model: there is always one, but there may be more if additional subsystems exist). These are measured directly by counting the appropriate nodes in the XML file defining the block diagram. Note that the model used for "Threshold Violation" has one additional block and line compared to the original model because an assertion block was added to instrument the model. The percentages show, for each model element counted, the amount of the original model deleted by slicing.

For the first and the third scenarios the reduction in total elements is around 50%, while for the the middle scenario "Division Overflow", the reduction is almost 80%. The percentages of deleted blocks are slightly lower, while the percentages for the deleted XML lines are slightly higher. This indicates that "average block complexity" is lower in the slice, which is in part because the slice retains some rather simple blocks in order to remain executable.

The numbers for the "Absolute Overflow Slice" and the "Threshold Violation Slice" are very similar, suggesting that the slices may be similar. However, a visual inspection revealed that the two slices are very different and share only 21 blocks (of the 44 blocks in the original model).

It turns out that each scenario generates additional warnings that indicate failures beyond those considered by Holling et al. [26]. In each scenario there are three such warnings. Therefore, we also applied XML-ORBS with a relaxed matching relation for each of the nine individual failures. The relaxed matching relation allows

Table 1: Slice size and percent reduction for the slices of the system studied by Holling et al. [27] using XML-ORBS with the test cases produced by 8Cage [26].

	Block Count		XML Lines		Total Elements	
Original Model	44		64		109	
Slices for all failures with strict matching relation						
Absolute Overflow Slice	28	36%	26	59%	55	50%
Division Overflow Slice	13	70%	9	86%	23	79%
Threshold Violation Slice	28	38%	28	57%	57	49%
Slices for specific failures with relaxed matching relation						
Absolute Overflow Slice (1)	25	43%	22	66%	48	56%
Absolute Overflow Slice (2)	26	41%	23	64%	50	54%
Absolute Overflow Slice (3)	27	39%	25	61%	53	51%
Division Overflow Slice (1)	11	75%	7	89%	19	83%
Division Overflow Slice (2)	12	73%	8	87%	21	81%
Division Overflow Slice (3)	13	70%	9	86%	23	79%
Threshold Violation Slice (1)	20	56%	19	71%	40	64%
Threshold Violation Slice (2)	12	73%	8	88%	21	81%
Threshold Violation Slice (3)	12	73%	9	86%	22	80%

the slice to generate fewer warnings but does not permit additional warnings. In addition, the specific failure of interest must be generated by the slice. For each of the nine failures, the slice is smaller, as shown in Table 1 under the heading "Slices for specific failures with relaxed matching relation." For the "Absolute Overflow" Scenarios, the reduction is slightly greater. The three different warnings generated in this scenario are all overflow warnings from a similar region of the model. Therefore the slices only differ slightly. The situation with the three "Division Overflow" Scenarios is similar: the three warnings are again from a common area of the model and thus yield similar reductions. It is interesting to note that the third slice is the same as the strict slice (the one that retains all three failures) indicating that it contains the other two as subslices. Finally, the three generated warnings in the "Threshold Violation" Scenario show greater difference because one of the warnings comes from a completely different area of the model. This leads to Threshold Violation Slice (1) being larger than and very different from Slices (2) and (3). All three slices are much smaller than the strict slice where use of the strict matching relation forces the slice to keep both areas of the model. Close inspection of slices (2) and (3) reveals that they are almost identical to the Division Overflow Slices (1) and (2).

Figure 2 shows the original model and the slices produced by XML-ORBS for the three specific failures considered by Holling et al. (to save space we have rearranged the layout of the original). The figure aims to show the general reductions of the model rather than focusing on detail. In each slice, the significant reduction in the number of model elements is visually evident. In particular, the complexity of the Division Overflow Slice is reduced to the point where an engineer can quickly comprehend the cause of the failure. Note that all the slices include blocks that are required to ensure the slice is executable. This accounts for the handful of isolated blocks seen in the figure. In addition, it is also visually evident that the

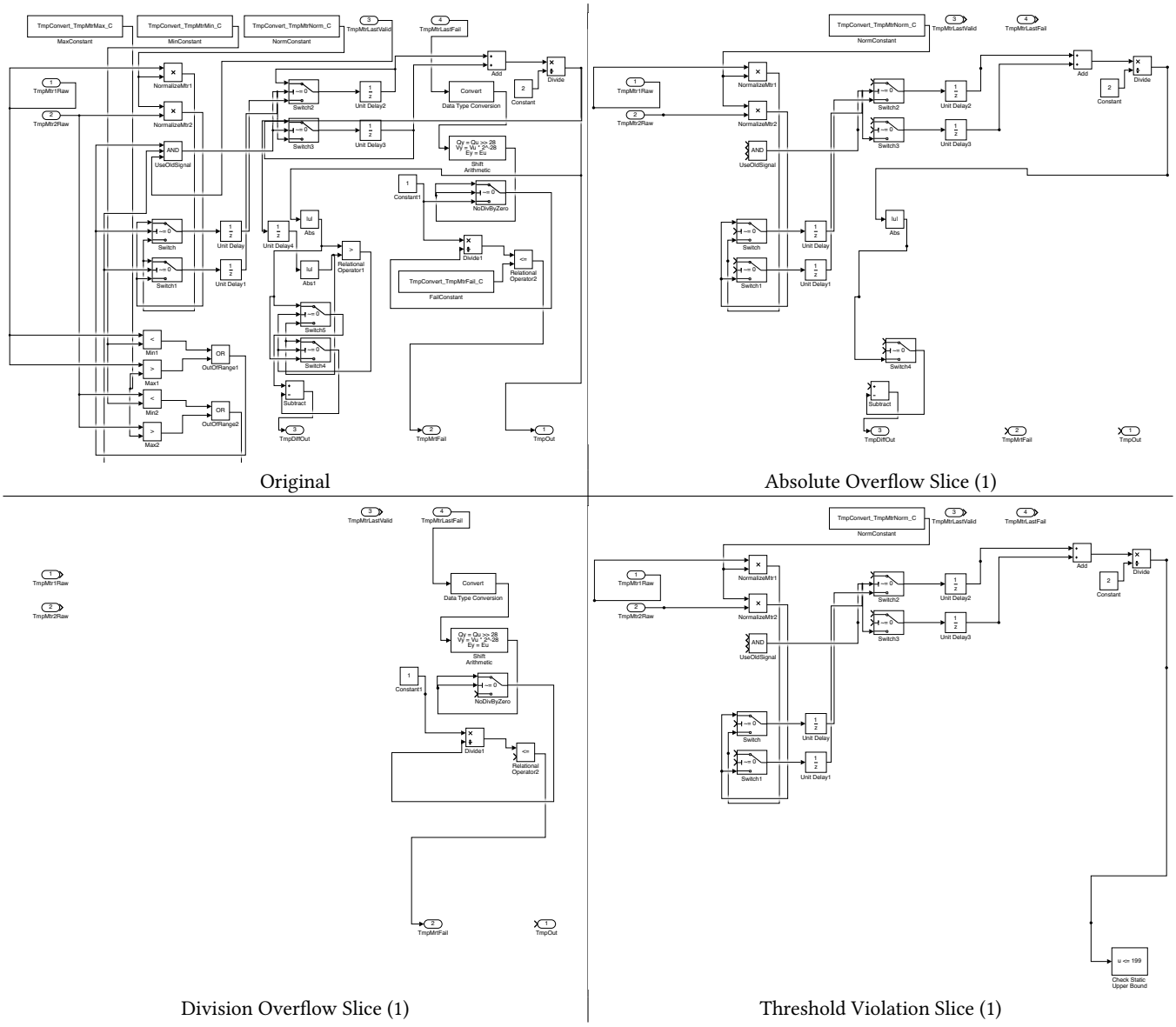

Figure 2: Slices for the example system of Holling et al. [27] using test cases generated by 8Cage [26].

Absolute Overflow Slice (1) and the Threshold Violation Slice (1) share a large common part of the model where 19 blocks are shared.

Figure 3 shows a more detailed illustration of part of the observational slice for the threshold violation criterion. It shows two switch blocks where XML-ORBS has removed the incoming trigger for both blocks (XML-ORBS also removed the upper input of Switch1). The trigger input determines the state of the switch. The incoming triggers act like the predicate of an if statement and cause a control dependence as they decide which incoming signal reaches the outgoing signal. The removal of these two inputs is a "red flag" to an engineer trying to diagnose the fault. In both cases, XML-ORBS was able to remove the incoming trigger because the failure is caused when the two switches are in their default states – it is errantly not necessary to flip the switches with a change of the incoming trigger. In comparison, a static slicer would clearly be unable to remove the incoming trigger due to the static dependence of the block's output on all three incoming signals. Moreover, a dependence-based dynamic slicer would not be able to remove the incoming trigger's connection because it computes the dynamic slice by removing from the static slice those dependences that are never executed. Alas, in this example, the dependences are executed and thus would be included in the dynamic slice. In short, the ability to remove executed, but unnecessary dependences allows XML-ORBS to remove large portions of models (such as the one shown in Figure 2) that a static or dynamic slicer could not.

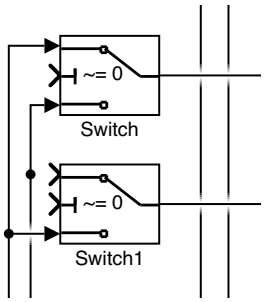

Figure 3: Detail of the Observational Slice for the Threshold Violation Criterion

6 EVALUATION

To evaluate XML-ORBS we undertook an empirical investigation using a range of models from various sources to determine how effective XML-ORBS is in slicing Simulink models and to consider the characteristics of the slicing process. The models used in the test corpus come from a variety of sources and domains and are of various sizes. They include realistic models and some small illustrative examples. Table 2 summarizes the models, their characteristics, the criteria used, and their type. Test cases were identified in various ways. For some models (e.g., *8Cage*) the test cases and criteria were identified by others (as described in Section 5), for others they were supplied with the model (e.g., the Campus Energy Modeling [3] library tests and demos), and the remainder were created with the model (e.g., for the set of illustrative models we created a model implementation of Danicic and Howroyd's Montreal Boat code [17]). In each case the output was either provided alongside the test case, or, where it was difficult to determine the outcome, the entire set of workspace variables produced was captured.

In each case, a model's suitability for inclusion in the corpus was determined by manual inspection to understand something of its operation (many of these models are in domains in which we do not have expertise), the data required to initiate simulation, and what might be considered important aspects of the resulting trajectory. Minor modifications (e.g., the inclusion of *ToWorkspace* blocks or assignment of values to workspace variables in start-up scripts) were made in a few cases to enable the automatic start-up and capture of model data and suppression of the interactive GUIs.

The models were also checked to ensure that they exhibited deterministic behavior under at least one of the two execution regimes available: the generic executer that starts a fresh MATLAB/Simulink session with each execution or the 'fast' executer that manages an ongoing MATLAB/Simulink instance as a subprocess, only restarting it when a crash or timeout occurs.

Finally, the experiments were undertaken using the XML-ORBS implementation described above on a MacBook Pro, 2.8 GHz Intel Core i7 with 16Gb RAM, SSD storage and running OS X Version 10.11.3. All models were run using MATLAB release 2015b in combination with Python scripts and instrumentation in MATLAB and XML-ORBS itself. Each model analysis was executed four times and the results (e.g., timings) averaged where appropriate.

6.1 Results and Discussion

This section presents the results from the test corpus (including those presented as part of the case study above). The true slice is an (undecidable) minimal slice. Slice size is the well-established and widely-accepted metric to measure slice precision. The case study has already shown how observational slicing is precise with respect to requiring those dependencies that are actually required to capture the slice. Here we consider the resulting slice sizes. This can be measured in a number of ways using the various model elements (blocks, lines, systems/subsystems, layers). Measurements can be made either via Simulink's metrics API (although this does not capture all the elements of interest such as the lines excluded, furthermore metric counts vary between MATLAB versions for the same model), or in terms of the raw XML. We present metrics as measured using XPath queries on the XML-represented block diagram.

Table 3 shows the original and sliced sizes of each model. For the 8Cage scenarios, Table 3 only shows the slices for all failures with the strict matching relation (Table 1 shows the other nine slices). The slice sizes are given as absolute sizes and as a proportion of the original program for each of the size metrics reported. In terms of total number of model elements, the sizes range from a slice that includes 83% of the original model ("Fourier synth, 0–3s") to slices than have removed almost every element ("Aeroradar"). The latter situation can occur when Simulink blocks have default values that are output in the absence of input. If the default value happens to coincide with the oracle, then the remainder of the model may be deleted and still produce the oracle trajectory.

Excluding the illustrative model executions leaves twenty realistic execution scenarios (the twelve 8Cage scenarios (Table 1), the three Mathworks' Simulink examples (Table 3), and the five models from the Campus Energy Modeling Project [3] (Table 3)). For almost half of these scenarios (nine of the twenty), the resulting slice contains less than 25% of the original model's elements. Note that it is not possible to directly compare the results of XML-ORBS to other static or dynamic slicers (even if they would be available to us) for two reasons. First, many of the criteria that we used cannot be mapped to traditional slicing criteria and second, XML-ORBS is the only Simulink slicer that guarantees that its slices are executable.

Execution characteristics are shown in Table 4. The table shows the number of iterations XML-ORBS does before no more deletions are possible, the number of times a model is executed after a deletion has been attempted, and the elapsed and CPU time needed.

It is interesting to note that in no case does the number of iterations exceed four, suggesting that there is limited dependence ordering that is not explicitly captured by the line connections (additional iterations are required where the deletion order differs from the dependence order, for example, a use must be deleted before the corresponding definition). Also, the majority of the models only need two iterations, which means that all possible deletions occur during the first iteration. The final iteration is, in one sense, superfluous in that it becomes final because nothing more can be deleted and thus it has no effect on the slice size. However, it must be completed to ensure that slicing is actually complete.

The results do not indicate any obvious correlation between size, execution time, type of observer, or number of executions

Table 2: Corpus of models used.

Model	Description	Source	Type	Test Cases	Criterion Type
Sum Product	Standard slicing example (e.g. [49])	Authors	C hand-translated into Simulink	Single (n=4)	Trajectory values
Montreal Boat	Standard slicing example (e.g. [17])	Authors	C hand-translated into Simulink	Single composite {(0, 0), (1, -1), (16, -1), (16, -19)}	Trajectory values
Delorder	Cross-block dependency example	Authors	C hand-translated into Simulink	Single (no params)	Trajectory values
Fourier synth	Fourier synthesis model	Authors	Simulink	Multiple (0-1, 1-2, 2-3, 0-3) seconds of audio output	Trajectory property (audio power at f Hz)
AeroRadar	Conceptual model of ATC radar	Mathworks' Simulink examples	Simulink	Default values	Trajectory values
AeroTrimlin	Autopilot control trimming				Traj. values & properties
Powerwindow	Car power window demonstration				Trajectory values
8Cage	Published Large Model [26, 27]	Holling et al. [27]	Simulink	From the 8Cage video [26]	Execution property (runtime warnings)
Constant Power	Power Source Tester	Campus Energy Modelling Project [3]	Simulink models supported by other libraries	Default values derived from the supplied test scripts	Assertions
EV	Electric vehicle tester				Assertions
EV Charging	Vehicle charging tester				Trajectory values
Weather	Weather testers				Trajectory values
PVWatts	PV Watts SSC co-simulation				Trajectory values

Table 3: Slice size results of XML-ORBS applied to the test corpus.

		Original Model				Sliced Model							
Model	Test Case	XML Block Count	XML Lines	XML Systems	Total Model Elements	XML Block Count		XML Lines		XML Systems		Total Model Elements	
Sum Product	n=4	20	20	2	42	14	70%	12	60%	2	100%	28	67%
Montreal Boat	Composite case	58	47	11	116	37	64%	21	45%	7	64%	65	56%
Delorder	Default	5	2	1	8	1	20%	0	0%	1	100%	2	25%
Fourier synth	0-1s	18	15	3	36	9	50%	7	47%	2	67%	18	50%
	1-2s	18	15	3	36	15	83%	12	80%	3	100%	30	83%
	2-3s	18	15	3	36	9	50%	7	47%	2	67%	18	50%
	0-3s	18	15	3	36	15	83%	12	80%	3	100%	30	83%
Aeroradar	Default values	129	134	11	274	1	1%	0	0%	1	9%	2	1%
AeroTrimlin	Default values	72	79	6	157	63	88%	61	77%	5	83%	129	82%
Powerwindow	Default values	238	200	27	465	53	22%	20	10%	7	26%	80	17%
8Cage	Absolute Overflow	44	64	1	109	28	64%	26	41%	1	100%	55	50%
	Division Overflow	44	64	1	109	13	30%	9	14%	1	100%	23	21%
	Threshold Violation	44	64	1	109	28	62%	28	43%	1	100%	57	51%
Constant Power	Single phase values	25	28	1	54	4	16%	0	0%	1	100%	5	9%
EV	Default values	6	7	1	14	5	83%	4	57%	1	100%	10	71%
EV Charging	Default values	34	46	1	81	14	41%	11	24%	1	100%	26	32%
Weather	Default values	5	5	1	11	3	60%	2	40%	1	100%	6	55%
PVWatts	Default values	37	45	2	84	14	38%	14	31%	1	50%	29	35%

required. The only correlation that can be seen is for 8Cage: in the Division Overflow Scenario, more is deleted with fewer attempted executions and in less time compared with the others.

Observing the implementation when running suggests that the proportion of crashes and timeouts increases in later iterations. This is to be expected as less of the model is deletable in each iteration.

While the current version of TreeORBS may not be suitable for interactive debugging (owing to long run-time), it could be integrated into overnight integration testing, automatically generating sliced models from failed tests for inspection the following day.

6.2 Stateflow

Existing slicers for Simulink do not fully handle the dependencies induced by and within Stateflow elements of a model. By working on the tree-structured XML, XML-ORBS is able to correctly and precisely slice Stateflow elements and the constituent state diagrams with respect to the test suite either alone, or as part of slicing the containing Simulink model. Since Simulink currently stores the Stateflow portions of the diagram in a separate file, one need only direct XML-ORBS to slice this file instead, or as well as, the main block diagram. The results shown in Table 3 sliced only the main

Table 4: Performance metrics resulting from applying XML-ORBS to the test corpus.

Model	Test Case	Iterations	Executions	Avg Elapsed Time (s)	Avg CPU Time (s)
Sum Product	n=4	2	88	378.52	389.75
Montreal Boat	Composite case	2	192	1455.41	1600.29
Delorder	Default	3	15	137.35	146.17
Fourier synth	0-1s	2	64	324.34	329.33
	1-2s	2	77	481.24	503.72
	2-3s	2	64	321.75	327.85
	0-3s	2	77	483.53	505.78
Aeroradar	Default values	2	335	8640.12	9209.62
AeroTrimlin	Default values	3	465	9577.37	10829.04
Powerwindow	Default values	2	604	3860.55	4132.47
8Cage	Absolute Overflow	3	274	1554.00	1655.5
	Division Overflow	3	185	500.67	506.21
	Threshold Violation	2	209	1060.33	1126.49
Constant Power	Single phase values	4	116	2728.49	1189.53
EV	Default values	2	36	1164.49	641.44
EV Charging	Default values	2	146	1163.58	645.33
Weather	Default values	2	26	732.37	407.68
PVWatts	Default values	2	155	7325.68	3343.51

block diagram, treating any Stateflow models present as simply a Simulink block. Table 5 shows slicing results for three models that contain Stateflow elements where the slicer was directed to slice both the Simulink and Stateflow files (two examples from the original test corpus and the other, 'Coin', developed by the authors specifically to evaluate Stateflow slicing). In two cases the slicer can remove the Stateflow entirely, in the other, it can reduce the size of the state machine. The slices arising from these analyses are thus more precise with respect to the test suite than those produced without slicing the Stateflow elements.

6.3 Threats to Validity

As with any study of dependence analysis techniques, the empirical results could be enhanced by further research on additional subjects. To avoid unnecessary external threats to the validity of the findings in terms of generalizability, we have drawn upon a variety of sources of Simulink models, including previously published work, and both large and small model sizes, for a variety of different domains. In terms of construct validity, minor changes have been made to models, in order to instrument these, as is standard with system analysis work. This instrumentation is entirely independent of the existing model computation, merely playing the role of data collection, allowing us to report our results.

7 RELATED WORK

Program slicing was introduced by Weiser [52] and developed substantially through the 1980s and 1990s particularly targeting static slicing. This culminated in industrial-tools [5] based on the widely-used System Dependence Graph (SDG) algorithm [28]. It remains a topic of interest to the present day [4, 44, 55].

Static program slicing produces a slice that is correct for all possible program executions (and thus has similarities with the cone-of-influence computation [24]). Dynamic slicing was introduced to tailor slices to a particular program execution [2, 31]. Most of these algorithms are based, at least in part, on static dependence analysis. For example, the first of Agrawal and Horgan's four dynamic slicing algorithms [2] removes from the static slice those elements not executed on a particular program execution.

Observation-based slicing was recently introduced as a form of dynamic slicing where the slice need only respect those dependencies actually *observed* [11]; a dependence is observed when its removal leads to the computation of different values at the slicing criterion. Basing slice computation purely on observation has far-reaching implications for the underlying algorithms. For example, a particular problem in dynamic slicing is caused by control dependence which must be pre-computed statically. Whenever a statement is included in a dynamic slice, all predicates on which the statement is control dependent are commonly included, together with all statements on which the predicate dynamically depends. Even when the predicate never changes its outcome, a dynamic slice tends to include it. An observational slice can remove the predicate and all statements on which only the predicate depends.

Observation-based slicing algorithms can also cater for different languages [11, 30, 54] and multilingual systems [11], whereas the white-box dependence analysis used by all previous slicing approaches forms a barrier to multilingual slicing. This combination of language independence and faithfulness to dependences actually observed during execution, has led to increased recent interest in multilingual [37] and observation based slicing techniques [29, 30].

Although our work is observational, it is also concerned with *model* slicing, which presents different challenges to the more widely studied paradigm of *program* slicing. Model slicing has become a recent topic of interest in its own right because of the importance and prevalence of software models [7]. Much of the work on model-based slicing has focused on UML models [6, 8, 34, 35]. In the remainder of this section we describe model-based slicing approaches that specifically target Simulink models, since these are most closely related to our own observational Simulink slicer.

There are three key challenges for Simulink dependence computation:

Table 5: Slice size results of XML-ORBS applied to combined Simulink/Stateflow examples.

		Original Model					Sliced Model									
Model	Test Case	Simulink XML Elements	XML States	XML Transitions	XML Data Nodes	Total Model Elements	Simulink XML Elements		XML States		XML Transitions		XML Data Nodes		Total Model Elements	
Aeroradar	Default	274	6	10	6	296	2	1%	0	0%	0	0%	0	0%	2	1%
Powerwindow	Default	465	3	5	2	475	80	17%	0	0%	0	0%	0	0%	80	17%
Coin	'Reject'	37	6	9	3	55	12	33%	1	0.2%	2	0.2%	3	100%	18	33%

(1) Because the model is a data-flow model, control-flow is implicit, rather than explicit as it is in programming languages, making the precise computation of control dependence non-trivial (see Reicherdt and Glesner [43]).
(2) There are *hidden* data dependences (beyond those represented by the signal lines that connect blocks) [22].
(3) Simulink models can include multiple embedded Stateflow models, that have a separate syntax and semantics; thus Simulink slicing is inherently multilingual.

Fortunately, the unique properties of observational slicing allow it to address all three of these challenges.

Most of the work on slicing Simulink models computes *static* slices and therefore is less suitable for applications like fault localization that require the precision of dynamic (or observational) slicing. Reicherdt and Glesner [43] introduced one of the earliest static Simulink slicers, based on Conditional Execution Contexts to capture control dependence, but it does not handle Stateflow models, with the result that it does not apply to many real-world Simulink models. Subsequently, Sridhar and Srinivasulu [45, 46] introduced a static Simulink slicer that does handle Stateflow models, but assumes the absence of parallel states, which also limits real-world applicability. Pantelic et al. were the first to incorporate, into a static Simulink slicer, the data dependences due to implicit signal flow involving data stores and Goto/From blocks [39]. More recently, Gerlitz and Kowalewski presented a flow-sensitive static slicer for Simulink models [22], but this also does not handle Stateflow models, limiting its real-world applicability.

The most closely related work to our observational Simulink slicer is that on dynamic Simulink slicers, of which only two exist. The dynamic Simulink slicer contained in the fault localization tool of Liu et al. [36] computes dynamic slices as the intersection of the static slice and the coverage information on executed elements. This approach mirrors the initial dynamic program slicing algorithm of Agrawal and Horgan [2], applied to Simulink models. Simply intersecting the static slice with the executed elements can lead to overly-large slices, as was also found by Agrawal and Horgan for dynamic program slicing, motivating them to develop more sophisticated dynamic program slicing algorithms. Furthermore, this approach computes so-called 'closure slices' [51], which may fail to compile. For Liu et al., closure slicing was sufficient for fault localization, but it is inadequate for many other slicing applications that require compilable or executable slices.

The second dynamic Simulink slicer is contained within the proprietary Simulink Design Verifier package slicing tool [48]. It can compute static Simulink slices by determining dependencies between blocks, signals, and model components. Moreover, the static slice can be *refined* (limited to the elements executed during a simulation). However, this is a commercial tool for which the algorithms used are not publicly available in the peer-reviewed literature. Nevertheless, while Simulink's slicer aims to produce executable slices, the documentation is quite clear that it makes no guarantees that the resulting slice will be executable. By its nature, XML-ORBS guarantees the production of executable slices.

In comparison with this previous work, our approach computes fully executable observational slices that are precise with respect to the chosen test suite, capture all implicit (hidden) dependencies dynamically traversed by the test suite, and slice the dependencies induced by Stateflow. Moreover, all previous slicing approaches restrict the slicing criteria and thus none offer the flexibility of the generalized observational slicing framework.

8 CONCLUSIONS AND FUTURE WORK

We have introduced the first observational slicing algorithm for models, using a tree-based approach that retains observation-based slicing's language independence. We applied our tree-oriented slicing algorithm to Simulink models, demonstrating the ability to significantly reduce model size. We evaluated the approach on nine real-world Simulink models, including models from previous publications and modeling projects in the public domain. In the evaluation, the resulting model had fewer than 25% of the original model's elements in 9 out of 20 scenarios (with a mean value of 35% of the original in the same 20 scenarios). Additionally, we presented three examples where Stateflow was explicitly sliced in combination with Simulink. Doing so led to even greater size reduction.

The case study presented in Section 5 demonstrates the utility of Tree-ORBS when combined with other approaches to identify observations of interest. The model size reductions are clear and substantial for the case study and other models.

Future work will include investigating heuristic approaches to reducing the computational cost, applying Tree-ORBS to other executable modeling languages, investigating applications of Tree-ORBS to reactive systems, parallelising the implementation, and investigating the effect of slicing on model complexity.

ACKNOWLEDGMENTS

We are very grateful to Holling, Pretschner, and Gemmar for kindly providing the model used in their paper [27] for our case study analysis. This work is partially supported by the UK Engineering and Physical Sciences Research Council under grant number EP/J017515/1.

Data from this paper is available at DOI: 10.14324/000.ds.1561328.

REFERENCES

[1] Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *33rd International Conference on Software Engineering*. 746–755.

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proc. of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*. 246–256.

[3] Alliance for Sustainable Energy. 2015. Campus Energy Modelling Project. (2015). github.com/NREL/CampusEnergyModeling

[4] Torben Amtoft and Anindya Banerjee. 2016. A Theory of Slicing for Probabilistic Control Flow Graphs. In *19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2016) (Lecture Notes in Computer Science)*, Bart Jacobs 0001 and Christof Löding (Eds.), Vol. 9634. 180–196.

[5] Paul Anderson and Tim Teitelbaum. 2001. Software Inspection Using CodeSurfer. In *Proc. of the 1st Workshop on Inspection in Software Engineering*. 4–11.

[6] Kelly Androutsopoulos, David Binkley, David Clark, Nicolas Gold, Mark Harman, Kevin Lano, and Zheng Li. 2011. Model Projection: Simplifying Models in Response to Restricting the Environment. In *Proceedings of the 33rd International Conference on Software Engineering*. 291–300. DOI: http://dx.doi.org/10.1145/1985793.1985834

[7] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurie Tratt. 2013. State-Based Model Slicing: A Survey. *Comput. Surveys* 45, 4, Article 53 (Aug. 2013), 36 pages.

[8] Kelly Androutsopoulos, David J. Clark, Mark Harman, Robert M. Hierons, Zheng Li, and Laurence Tratt. 2013. Amorphous Slicing of Extended Finite State Machines. *IEEE Transactions on Software Engineering* 39, 7 (July 2013).

[9] Jon Beck and David Eichmann. 1993. Program and interface slicing for reverse engineering. In *Proc. of the 15th International Conference on Software Engineering (ICSE)*. 509–518.

[10] David Binkley. 1998. The application of program slicing to regression testing. *Information and Software Technology* 40, 11 and 12 (1998), 583–594.

[11] David Binkley, Nicolas Gold, M. Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-Independent Program Slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. 109–120.

[12] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2015. ORBS and the limits of static slicing. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–10.

[13] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2017. Tree-Oriented vs. Line-Oriented Observation-Based Slicing. In *2017 IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Under Review.

[14] Cristina Cifuentes and Antoine Fraboulet. 1997. Intraprocedural Static Slicing of Binary Executables. In *Proc. of the International Conference on Software Maintenance (ICSM)*. 188–195.

[15] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. 1995. Identifying reusable functions using specification driven program slicing: a case study. In *Proc. of the International Conference on Software Maintenance (ICSM)* (Nice, France). 124–133.

[16] M. Collard. 2005. Addressing Source Code Using srcML. In *IEEE International Workshop on Program Comprehension Working Session (IWPC'05)*.

[17] Sebastian Danicic and John Howroyd. 2002. Montréal Boat Example. Source Code Analysis and Manipulation (SCAM 2002) conference resources website. (2002). http://www.ieee-scam.org/2002/Slides_ct.html

[18] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. 1996. Understanding function behaviours through program slicing. In 4^{th} *International Workshop on Program Comprehension* (Berlin, Germany). 9–18.

[19] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. 2008. Clone Detection in Automotive Model-based Development. In *Proceedings of the 30th International Conference on Software Engineering*. 603–612. DOI: http://dx.doi.org/10.1145/1368088.1368172

[20] Ran Ettinger and Mathieu Verbaere. 2004. Untangling: a slice extraction refactoring. In *Proc. of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*. 93–101. DOI: http://dx.doi.org/10.1145/976270.976283

[21] Keith B. Gallagher and James R. Lyle. 1991. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* 17, 8 (1991), 751–761.

[22] Thomas Gerlitz and Stefan Kowalewski. 2016. Flow Sensitive Slicing for MATLAB/Simulink Models. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA 2016)*. 81–90.

[23] Ákos Hajnal and István Forgács. 2011. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software: Evolution and Process* 24, 1 (2011), 67–82.

[24] Klaus Havelund and Jens Ulrik Skakkebæk. 1999. *Applying Model Checking in Java Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 216–231. DOI: http://dx.doi.org/10.1007/3-540-48234-2_17

[25] Robert Mark Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. 2002. Conditioned Slicing Supports Partition Testing. *Software Testing, Verification and Reliability* 12 (March 2002), 23–28.

[26] Dominik Holling, Alexander Pretschner, and Matthias Gemmar. 2014. 8Cage Demo Video. (Aug. 2014). https://www.youtube.com/watch?v=1xmA4fHVi4Y Accessed 20 July 2016.

[27] Dominik Holling, Alexander Pretschner, and Matthias Gemmar. 2014. 8Cage: Lightweight Fault-based Test Generation for Simulink. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 859–862.

[28] Susan Horwitz, Thomas Reps, and David Wendell Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM transactions on programming languages and systems* 12, 1 (1990), 26–61.

[29] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *25th International Joint Conference on Artificial Intelligence*.

[30] David Insa, Sergio Pérez, and Josep Silva. 2016. How to construct a suite of program slices. In *16th Jornadas sobre Programación y Lenguajes (PROLE 2016)*.

[31] Bogdan Korel and Janusz Laski. 1988. Dynamic Program Slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163.

[32] Bogdan Korel and Jurgen Rilling. 1997. Dynamic Program Slicing in Understanding of Program Execution. In *Proc. of the* 5^{th} *International Workshop on Program Comprehension (IWPC)*. 80–89.

[33] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. 2002. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering* 7 (2002), 49–76.

[34] J.T. Lallchandani and R. Mall. 2010. Integrated state-based dynamic slicing technique for UML models. *IET Software* 4 (February 2010), 55–78(23). Issue 1.

[35] Jaiprakash T. Lallchandani and R. Mall. 2011. A Dynamic Slicing Technique for UML Architectural Models. *IEEE Transactions on Software Engineering* 37, 6 (2011), 737–771. DOI: http://dx.doi.org/10.1109/TSE.2010.112

[36] Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability* 26, 6 (2016), 431–459. DOI: http://dx.doi.org/10.1002/stvr.1605 stvr.1605.

[37] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2015. Cross-language program slicing for dynamic web applications. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 369–380.

[38] Marta Olszewska (pląska). 2011. *Simulink-Specific Design Quality Metrics*. Technical Report 1002. Turku Centre for Computer Science.

[39] Vera Pantelic, Steven Postma, Mark Stephen Lawford, and Marc Bender. 2015. A Toolset for Simulink: Improving Software Engineering Practices in Development with Simulink. In *Proceedings of 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015)*.

[40] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. 2009. Model Based Analysis and Test Generation for Flight Software. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*. 83–90. DOI: http://dx.doi.org/10.1109/SMC-IT.2009.18

[41] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Complete and Accurate Clone Detection in Graph-based Models. In *Proceedings of the 31st International Conference on Software Engineering*. 276–286. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070528

[42] Alexander Pretschner. 2017. 8Cage: Lightweight Fault-Based Test Generation for Simulink. (2017). www22.in.tum.de/en/tools/8Cage/ Accessed 4 July 2017.

[43] Robert Reicherdt and Sabine Glesner. 2012. Slicing MATLAB Simulink Models. In *Proceedings of the 34th International Conference on Software Engineering*. 551–561.

[44] Josep Silva. 2012. A Vocabulary of Program Slicing-Based Techniques. *Comput. Surveys* 44, 3 (June 2012), 12:1 – 12:48.

[45] Adepu Sridhar. 2013. *Generating Test Sequences and Slices for Simulink/Stateflow Models*. Master's thesis. National Institute of Technology, Rourkela, India.

[46] Adepu Sridhar and D. Srinivasulu. 2014. Slicing MATLAB Simulink/Stateflow Models. In *Intelligent Computing, Networking, and Informatics: Proceedings of the International Conference on Advanced Computing, Networking, and Informatics, India, June 2013*. Springer India, 737–743. DOI: http://dx.doi.org/10.1007/978-81-322-1665-0_74

[47] The Mathworks Inc. 2016. Simulink. (2016). http://uk.mathworks.com/products/simulink/ Accessed 21 July 2016.

[48] The Mathworks Inc. 2016. Simulink Design Verifier: Isolating Problematic Behaviour with Model Slicer. (2016). https://uk.mathworks.com/products/sldesignverifier/features.html#isolating-problematic-behavior-with-model-slicer Accessed 20 July 2016.

[49] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3 (1995), 121–189.

[50] Paolo Tonella. 2003. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Transactions on Software Engineering* 29, 6 (2003), 495–509.

[51] Guda A. Venkatesh. 1991. The semantic approach to program slicing. In *Programming Language Design and Implementation (PLDI 1991)*. 26–28.
[52] Mark Weiser. 1979. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* Ph.D. Dissertation. University of Michigan, Ann Arbor, MI.
[53] Mark Weiser and Jim Lyle. 1985. Experiments on slicing-based debugging aids. In *Empirical Studies of Programmers: First Workshop*. 187–197.
[54] Shin Yoo, David Binkley, and Roger Eastman. 2014. Seeing Is Slicing: Observation Based Slicing of Picture Description Languages. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 175–184. DOI : http://dx.doi.org/10.1109/SCAM.2014.26
[55] Yiji Zhang and Raúl A. Santelices. 2016. Prioritized static slicing and its application to fault localization. *Journal of Systems and Software* 114 (2016), 38–53.