Abstract

Model-driven software engineering raises abstraction levels 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 (observational slicing) 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 in order to reduce them to the part of the models responsible for an observed failure, a classic slicing criterion, or the part of the model that is actually tested by a test, which is not possible with any previous slicing approach. 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.
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 [1–3], fault localization [3], impact analysis [4], slicing [5,6], and clone detection [7,8]; 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 (or fault) of interest. Such techniques help uncover the causes of the inevitable errors found when modeling complex systems. Example model slicing techniques include those based on dependence analysis [6] and those based on model projection [5].

This paper presents a new method, Tree Observational Slicing (Tree-ORBS), for dynamically slicing models where the only pre-requisite is that the model is represented using a tree-structured representation (e.g., XML). Tree-ORBS is based on ORBS [9–11], 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 generated by the run-time environment rather than the model).

One widely used product for modeling that uses a tree structured representation is Mathworks’ Simulink [12], 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. It can source and sink data from files or the underlying MATLAB workspaces. Simulations can be run from the GUI or a MATLAB script. Constructing a model broadly involves placing blocks on a canvas, setting their properties, connecting them with lines that control the flow of data, and setting global model properties. Simulink models have been the subject of research for various model analysis techniques including test-case generation and fault localization [1–3], clone detection [7,8], and quality assessment metrics [13].

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 [3]. 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

- A generalization of observation-based slicing (observational slicing) allowing different types of observation and criteria.
- An algorithm, Tree-ORBS, for observational slicing of tree-structured representations.
- An implementation of a Tree-ORBS slicer for XML (XML-ORBS).
- A case study that demonstrates the utility of Tree-ORBS by achieving 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.

The remainder of the text is organized thus: Section 2 introduces slicing, ORBS, and the application of ORBS to models. Section 3 presents a generalization of observation-based slicing, the basis of the Tree-ORBS algorithm presented in Section 4 and its application to Simulink models. A case study (Section 5) and empirical investigation of XML-ORBS on various Simulink models is then presented and discussed (Section 6). Finally, related work is presented (Section 7) and the paper concludes (Section 8).

2 Background

2.1 Slicing Programs and Models

Program slicing is a technique for deleting parts of a software system that are irrelevant to some chosen slicing criterion [14–16]. There are many forms of slicing including static, dynamic, quasi-static, conditioned, amorphous and syntax-preserving [15]. Because of the way it simplifies the system by focusing only on the slicing criterion, slicing has found many software engineering applications, including testing [17,18], debugging [19,20], maintenance [21,22], re-engineering [23], re-use [24,25], comprehension [26–28] and refactoring [29].

The present paper is inspired by a recently-introduced form of dynamic slicing known as Observation-Based Slicing [9–11], which dynamically slices multi-language 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 speculative 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 element is 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) [11]. Observation-based slicing produces inherently smaller sizes than static slicing, because it is based on dynamic observation, but it can also produce smaller sizes than traditional dynamic approaches too, because of its focus on observation (rather than dynamically traversed statically-defined dependence) [9,30].

One important feature of ORBS, an implementation of observation-based slicing [9], is the way in which focusing on deleting lines instead of statements...
liberates the slicing algorithm from the need for detailed (and thereby expensive and brittle) semantic analysis; lines can be deleted and their effects simply observed, a property from which ORBS derives its language independence. Furthermore, using only speculative lexical deletion is highly language-independent, and thereby ORBS is able to slice systems expressed in multiple languages. In contrast, such multi-lingual 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, is visually represented as a diagram with functional blocks (possibly encapsulating sub-systems) connected by lines representing the flow of data during simulation, which we treat as the model’s ‘source code’. Associated with the model are a range of parameters governing the way the simulation is run and the model may interact with external devices, files, and the underlying MATLAB workspace.

There are several existing approaches to slicing Simulink models [6, 31–33], 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 [9] could be applied to simply delete XML lines. However, this approach would clearly be suboptimal, since it would fail to take advantage of the tree-based nature of the XML notation, in which whole sub-trees can be pruned away, rather than individual lines.

Without sacrificing language independence, but at the same time exploiting the tree structure of XML, we introduce a tree-based observational slicing algorithm, Tree-ORBS. We then go on to implement the algorithm for Simulink models and evaluate the resulting implementation on a collection of real-world Simulink models. Unlike non-observational alternatives, the resulting slices, which are considerably smaller than the original models, are fully executable. Thus they can be loaded into MATLAB/Simulink and used in place of the original.

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) for which variables and statements are well defined. To adapt to other programming languages and other 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 and is defined as follows:
(Trajectory) Observation-Based Slice [9]: An observation-based slice $S$ of a program $P$ on a slicing criterion $C = (v, l, I)$ composed of variable $v$, line $l$, and set of inputs $I$, is any executable program with the following properties:

1. The execution of $P$ for every input $I$ in $I$ 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 $I$ halts and produces a sequence of values $V(S, I, v, l)$ for variable $v$ at line $l$.
4. $\forall I \in I \ V(P, I, v, l) = V(S, I, v, l)$.

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 (referred to as 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 $C = (O, R, I)$ composed of an observer $O$, a matching relation $R$, and a set of inputs $I$, is any executable program with the following properties:

1. The execution of $P$ for every input $I$ in $I$ 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 $I$ halts and produces the observed behavior $O(S, I)$.
4. $\forall I \in I \ O(S, I) \sim_R O(P, I)$.

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 produced by program $P$ when $P$ is run on each input $I \in I$. If the matching relation, $R$, is equality, then the corresponding generalized observational slice $S$ is $P$ after dead code is removed (with respect to the inputs of $I$).

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 $C = (v, l, I)$ 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 $T$ where each test case $T \in T$ 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 $I = \{T \mid T \in T\}$ 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 $R = \{(\text{PASS, PASS}), (\text{FAIL, FAIL}), (\text{FAIL, PASS})\}$.

A similar generalization extended the ORBS algorithm to slice picture description languages [10] 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 of 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 $I$. For example, if $I$ 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 first presents our algorithm for slicing models and then describes some of the details from 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 \( I \). 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: \texttt{delete} removes the subtree rooted at component \( c \) from the tree representation of model \( M \). \texttt{children} returns the children of component \( c \). Finally, \texttt{append}, \texttt{dequeue}, and \texttt{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. 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.

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 shell-script to set up projects for analysis and a core Python script to 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 \( I \in I \) includes environmental inputs and execution settings.

Applying XML-ORBS to Simulink models requires some additional configuration to enable XML-ORBS to start MATLAB/Simulink as well as and executing models, and retrieving trajectories. An \texttt{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 \texttt{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’ script-triggered version because of the computational cost of starting MATLAB/Simulink, some models were observed to have non-deterministic behavior.
 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** $I \in I$
2. $V_i \leftarrow O(M, I)$
3. **repeat**
4. $\text{nothingDeleted} \leftarrow \text{True}$
5. $q \leftarrow N$
6. **while** $\neg \text{EMPTY}(q)$
7. $c \leftarrow \text{DEQUEUE}(q)$
8. $M' \leftarrow \text{DELETE}(M, c)$
9. $\text{deletable} \leftarrow \text{True}$
10. **foreach** $I \in I$
11. $V'' \leftarrow O(M', I)$
12. **if** $V_i \not\sim_R V''$
13. $\text{deletable} \leftarrow \text{False}$
14. **if** $\text{deletable}$
15. $M \leftarrow M'$
16. $\text{nothingDeleted} \leftarrow \text{False}$
17. **else**
18. $q \leftarrow \text{APPEND}(q, \text{CHILDREN}(c))$
19. **until** $\text{nothingDeleted}$
20. **return** $M$

Figure 1: Tree-ORBS Algorithm
under this executer despite having deterministic behavior when used with the
generic executer (perhaps due to some internal persistent state within Simulink
between runs).

Recall that the generalized framework requires that model evaluation termin-
inate for all \( I \in I \). 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 [3]. 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 accom-
panying their paper [34]). They identify three faults in the model, an absolute
value overflow, a division overflow, and a threshold violation, and generate in-
puts to trigger these faults. In the following, we demonstrate how our approach
is used to reduce the \( 8Cage \) model to the sub-model responsible for the fail-
ures. Since this is an experiment relying on execution properties (the induction
of failures), XML-ORBS is configured to use an observer that extracts warn-
ings 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.” \( \text{Block Count} \) and \( \text{XML Lines} \) refer to the number
of elements of the diagram (model) remaining (the \( \text{Total Elements} \) 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 al-
most 80%. The percentages of deleted blocks are slightly lower, but the percent-
ages 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
Table 1: Slice size and percent reduction for the slices of the system studied by Holling et al. [3] using XML-ORBS with the test cases produced by 8Cage [34].

<table>
<thead>
<tr>
<th>Block Count</th>
<th>XML Lines</th>
<th>Total Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Model</td>
<td>44</td>
<td>64</td>
</tr>
</tbody>
</table>

**Slices for all failures, with strict matching relation**

- Absolute Overflow Slice: 28 blocks, 36% reduction (26 lines, 59% reduction (55 elements, 50% reduction)
- Division Overflow Slice: 13 blocks, 70% reduction (9 lines, 86% reduction (23 elements, 79% reduction)
- Threshold Violation Slice: 28 blocks, 38% reduction (28 lines, 57% reduction (57 elements, 49% reduction)

**Slices for specific failures, with relaxed matching relation**

- Absolute Overflow Slice (1): 25 blocks, 43% reduction (22 lines, 66% reduction (48 elements, 56% reduction)
- Absolute Overflow Slice (2): 26 blocks, 41% reduction (23 lines, 64% reduction (50 elements, 54% reduction)
- Absolute Overflow Slice (3): 27 blocks, 39% reduction (25 lines, 61% reduction (53 elements, 51% reduction)
- Division Overflow Slice (1): 11 blocks, 75% reduction (7 lines, 89% reduction (19 elements, 83% reduction)
- Division Overflow Slice (2): 12 blocks, 73% reduction (8 lines, 87% reduction (21 elements, 81% reduction)
- Division Overflow Slice (3): 13 blocks, 70% reduction (9 lines, 86% reduction (23 elements, 79% reduction)
- Threshold Violation Slice (1): 20 blocks, 56% reduction (19 lines, 71% reduction (40 elements, 64% reduction)
- Threshold Violation Slice (2): 12 blocks, 73% reduction (8 lines, 88% reduction (21 elements, 81% reduction)
- Threshold Violation Slice (3): 12 blocks, 73% reduction (9 lines, 86% reduction (22 elements, 80% reduction)

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. [34]. 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 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. A 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 three 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 and the figure aims to show the
general reduction of the model in each case rather than focusing on detail.) In each of the three slices, 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 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 will not be able to remove the incoming triggers connection. This is because it
computes the dynamic slice by removing from the static slice those dependences that are never executed. However, 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 would include.

6 Evaluation

To evaluate XML-ORBS we undertake 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., Cage) the test cases and criteria were identified by others (i.e., Section 5), for others they were supplied with the model (e.g., the Campus Energy Modeling [35] library tests and demos), and the remainder were either created with the model (e.g., the set of illustrative models we created ourselves like the Montreal Boat example [36]). 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 as well as to suppress 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: either the generic executor that starts a fresh MATLAB/Simulink session with each execution or the ‘fast’ executor 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 effectiveness of any slicer is primarily considered in terms of the size of the resulting slices. The case study has already shown how observational slicing is precise with respect to 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 are excluded, and 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”).

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 [35] (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 time needed (both elapsed time and CPU time).

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 presented do not indicate any obvious correlation between size, execution time, type of observer, or number of executions required. The only correlation that can be extracted is for the 8Cage scenarios: in the Division Overflow Scenario, more is deleted, with fewer attempted executions, and in less time, compared to the other two scenarios.

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

6.2 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 has a long heritage dating back to the seminal work of Mark Weiser, who introduced program slicing as part of his PhD thesis [16]. Throughout the 1980s and 1990s there was much algorithmic development, particularly targeting static program slicing, culminating in the production of industrial tools such as Grammatech’s CodeSurfer [38], based on the widely-used System Dependence Graph (SDG) algorithm [39]. Program slicing remains a topic of interest and development to the present day [40,41], and there is a recent survey of terminology by Silva [15].

Traditional program slicing was static, producing a slice that was correct for possible program executions. Dynamic slicing was introduced to tailor slices to a particular program execution [42,43]. However, most of these algorithms are still based on static dependence analysis. For example, the most naive approach simply removes from the static slice those elements that are not executed on the particular program execution. This approach dates back to Agrawal and Horgan who first considered computing dynamic slices using a dependence graph. This particular approach is the first (and simplest) of their four dynamic slicing algorithms [42].

More recently, observation-based slicing was introduced as a form of dynamic slicing in which the slice need only respect those dependencies actually observed [9]; 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, however, can remove the predicate and all statements on which only the predicate depends.

As another example, observation-based slicing algorithms can not only cater for different languages [9,10,30], but also for multilingual systems [9], 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 [44] and observation based slicing techniques [30,45].

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 [14]. Much of the work on model based slicing has focused on UML models [5,46–48]. 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:

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 [6].

2. There are hidden data dependences (beyond those represented by the signal lines that connect blocks) [31].

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, which have hitherto remained unaddressed in all existing Simulink slicing approaches, none of which is observational.

Most of the work on slicing Simulink models computes static slices and therefore is not suitable for application scenarios like fault localization that require the precision of dynamic (or observational) slicing. Reicherdt and Glesner [6] 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 [2,32] 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 [33]. More recently, Gerlitz and Kowalewski presented a flow-sensitive static slicer for Simulink models [31], but this also does not handle Stateflow models, limiting its real-world applicability.

The most closely related work to our observational Simulink slicer are dynamic Simulink slicers, of which only two exist. The dynamic Simulink slicer
contained in the fault localization tool of Liu et al. [49] computes dynamic slices as the intersection of the static slice and the coverage information on executed elements. This approach mirrors the initial program slicing algorithm of Agrawal and Horgan [42], applied to Simulink models. Simply intersecting the static slice with the executed elements is a naive approach that 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’ [50], 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 [51]. 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. In contrast, by its very 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 respect the dependencies induced by Stateflow in the model. Moreover, all previous slicing approaches restrict the slicing criteria and thus none offer the flexibility that the generalized observational slicing framework provides.

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, expressed as XML trees, 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.

The case study presented in this paper further demonstrates the utility of Tree-ORBS, particularly when combined with other approaches that can identify observations of interest and ways to trigger them. The model size reductions are clear and substantial, both for the case study, and in the broader set of models and test cases.

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

We are very grateful to Holling, Pretschner, and Gemmar for kindly providing the model used in their paper [3] for our case study analysis.

References


Table 2: Corpus of models used.

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
<th>Source</th>
<th>Type</th>
<th>Test Cases</th>
<th>Criterion Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum Product</td>
<td>Standard slicing example (e.g. [37])</td>
<td>Authors</td>
<td>C hand-translated into Simulink</td>
<td>Single (n=4)</td>
<td>Trajectory values</td>
</tr>
<tr>
<td>Montreal Boat</td>
<td>Standard slicing example (e.g. [36])</td>
<td>Authors</td>
<td>C hand-translated into Simulink</td>
<td>Single composite {(0, 0), (1, -1), (16, -1), (16, -19)}</td>
<td>Trajectory values</td>
</tr>
<tr>
<td>Delorder</td>
<td>Cross-block dependency example</td>
<td>Authors</td>
<td>C hand-translated into Simulink</td>
<td>Single (no params)</td>
<td>Trajectory values</td>
</tr>
<tr>
<td>Fourier synth</td>
<td>Fourier synthesis model</td>
<td>Authors</td>
<td>Simulink</td>
<td>Multiple (0-1, 1-2, 2-3, 0-3) seconds of audio output</td>
<td>Trajectory property (audio power at f Hz)</td>
</tr>
<tr>
<td>AeroRadar</td>
<td>Conceptual model of ATC radar</td>
<td>Mathworks' Simulink examples</td>
<td>Simulink</td>
<td>Default values</td>
<td>Trajectory values</td>
</tr>
<tr>
<td>Aero/Trimlin</td>
<td>Autopilot control trimming</td>
<td>Simulink</td>
<td></td>
<td></td>
<td>Traj. values &amp; properties</td>
</tr>
<tr>
<td>Powerwindow</td>
<td>Car power window demonstration</td>
<td>Simulink</td>
<td></td>
<td></td>
<td>Trajectory values</td>
</tr>
<tr>
<td>8Cage</td>
<td>Published Large Model [3,34]</td>
<td>Holling et al. [3]</td>
<td>Simulink</td>
<td>From the 8Cage video [34]</td>
<td>Execution property (runtime warnings)</td>
</tr>
<tr>
<td>Constant Power</td>
<td>Power Source Tester</td>
<td>Campus Energy Modelling Project [35]</td>
<td>Simulink</td>
<td>Default values derived from the supplied test scripts</td>
<td>Assertions</td>
</tr>
<tr>
<td>EV</td>
<td>Electric vehicle tester</td>
<td>Simulink</td>
<td></td>
<td></td>
<td>Assertions</td>
</tr>
<tr>
<td>EV Charging</td>
<td>Vehicle charging tester</td>
<td>Simulink</td>
<td></td>
<td></td>
<td>Trajectory values</td>
</tr>
<tr>
<td>Weather</td>
<td>Weather testers</td>
<td>Simulink</td>
<td></td>
<td></td>
<td>Trajectory values</td>
</tr>
<tr>
<td>PVWatts</td>
<td>PV Watts SSC co-simulation</td>
<td>Simulink</td>
<td></td>
<td></td>
<td>Trajectory values</td>
</tr>
</tbody>
</table>
Table 3: Slice size results of XML-ORBS applied to the test corpus.

<table>
<thead>
<tr>
<th>Model</th>
<th>Test Case</th>
<th>Original Model</th>
<th>Sliced Model</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>XML Block Count</td>
<td>XML Lines</td>
<td>XML Systems</td>
<td>Total Model Elements</td>
<td>XML Block Count</td>
</tr>
<tr>
<td>Sum Product</td>
<td>n=4</td>
<td>20</td>
<td>20</td>
<td>2</td>
<td>42</td>
<td>14</td>
</tr>
<tr>
<td>Montreal Boat</td>
<td>Composite case</td>
<td>58</td>
<td>47</td>
<td>11</td>
<td>116</td>
<td>37</td>
</tr>
<tr>
<td>Delorder</td>
<td>Default</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Fourier synth</td>
<td>0-1s</td>
<td>18</td>
<td>15</td>
<td>3</td>
<td>36</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1-2s</td>
<td>18</td>
<td>15</td>
<td>3</td>
<td>36</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2-3s</td>
<td>18</td>
<td>15</td>
<td>3</td>
<td>36</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>0-3s</td>
<td>18</td>
<td>15</td>
<td>3</td>
<td>36</td>
<td>15</td>
</tr>
<tr>
<td>Aeroradar</td>
<td>Default values</td>
<td>129</td>
<td>134</td>
<td>11</td>
<td>274</td>
<td>1</td>
</tr>
<tr>
<td>AeroTrimlin</td>
<td>Default values</td>
<td>72</td>
<td>79</td>
<td>6</td>
<td>157</td>
<td>63</td>
</tr>
<tr>
<td>Powerwindow</td>
<td>Default values</td>
<td>238</td>
<td>200</td>
<td>27</td>
<td>465</td>
<td>53</td>
</tr>
<tr>
<td>8Cage</td>
<td>Absolute Overflow</td>
<td>44</td>
<td>64</td>
<td>1</td>
<td>109</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Division Overflow</td>
<td>44</td>
<td>64</td>
<td>1</td>
<td>109</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Threshold Violation</td>
<td>44</td>
<td>64</td>
<td>1</td>
<td>109</td>
<td>28</td>
</tr>
<tr>
<td>Constant Power</td>
<td>Single phase values</td>
<td>25</td>
<td>28</td>
<td>1</td>
<td>54</td>
<td>4</td>
</tr>
<tr>
<td>EV</td>
<td>Default values</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>EV Charging</td>
<td>Default values</td>
<td>34</td>
<td>46</td>
<td>1</td>
<td>81</td>
<td>14</td>
</tr>
<tr>
<td>Weather</td>
<td>Default values</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>PVWatts</td>
<td>Default values</td>
<td>37</td>
<td>45</td>
<td>2</td>
<td>84</td>
<td>14</td>
</tr>
</tbody>
</table>
Table 4: Performance metrics resulting from applying XML-ORBS to the test corpus.

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