JOT
JOURNAL OF
OBJECT TECHNOLOGY

The OSATE Slicer: Graph-Based Reachability for Architectural Models

Sam Procter
Software Engineering Institute, Carnegie Mellon University, United States of America

ABSTRACT Model-based specification of embedded, critical systems (i.e., systems whose failure is deemed unacceptable) is increasingly becoming the standard of practice. However, analysis of these models can prove challenging when the models grow in size and complexity, which is common as more types and volume of data are loaded into them. One well-studied technique for grappling with complexity in models is *slicing*, where slices of models are highlighted according to some criterion. In this work, we describe a new software tool: the *OSATE Slicer*, which extends the concept of slicing to architectural models of embedded, critical systems. It does this by calculating of various notions of reachability which can be used to support both manual and automated analyses of system models. We then evaluate the utility of our approach based on several existing analyses and its performance based on a pre-existing corpus of architectural models and find both to be satisfactory.

KEYWORDS Architecture Analysis and Design Language, Model Based Engineering, Slicing, Reachability, Model Analysis, Safety, Security

1. Introduction

Model-based engineering (MBE) is an increasingly popular way to build a range of systems, including *critical systems*, i.e., systems where failure would cause injury, death, or unacceptable financial losses. While the range of critical systems is broad, common examples include medical devices, power-generation systems (in particular those which rely on nuclear power), and all manner of high-speed transportation such as aircraft, automobiles, and trains. The core idea of MBE is to construct a model of a system which hides some implementation details in order to allow human users (e.g., modelers, safety or security analysts, certification authorities) or automated analyses to calculate properties of the final system without requiring access to it. Such analyses can be run early in the system development lifecycle, even before the modeled component or full system is built. Many of these analyses rely on notions of *reachability* between different system elements, but calculating reachability can be error-prone and a source of significant performance degradation in MBE tooling. This paper describes recent work towards our goal of supporting accurate, maintainable, and performant reachability calculations on models of critical systems.

JOT reference format:
Sam Procter. *The OSATE Slicer: Graph-Based Reachability for Architectural Models*. Journal of Object Technology. Vol. 22, No. 2, 2023.
 http://dx.doi.org/10.5381/jot.2023.22.2.a3

1.1. A Single Source of Truth

The *Architecture Analysis and Design Language* (AADL) (*SAE Architecture Analysis and Design Language (AADL)* 2022) is a language used for MBE, particularly of embedded, critical systems. The *Open Source AADL Tool Environment* (OSATE) is an integrated development environment that supports editing and analyzing architecture models specified in AADL. One of the selling points of AADL, and indeed MBE more generally, is that a range of different system domains (e.g., safety, security, performance) can be modeled in a single specification; this provides a number of advantages over a siloed approach where different stakeholders develop and analyze domain-specific models. This *single source of truth* approach (Feiler et al. 2009) can help different stakeholders stay synchronized, reduce integration issues, and reduce overall modeling effort. However, a gap remains between the potential power and ease of having multiple analyses operate on a single input model and its realization.

Easy traversal of a model and identification of data flows through a system would reduce this gap considerably. In particular, we advocate for the use of efficient *slicing* in general, and *reachability* calculations in particular, in addressing two primary reasons for the gap between the single source of truth

concept's potential and its realization: the cognitive burden of multiple domain-specific annotations on a single model and the challenge of repeatedly traversing large models.

1.1.1. Cognitive Burden

Architecture models extended with domain-specific annotations are, as a truism, likely to be useful to analysts and certification authorities from that specific domain. That utility decreases, or at least is more difficult to fully and easily realize, when annotations from multiple domains are combined into a single model. While required for the single source of truth approach, multi-domain models can be larger and more challenging to understand than those focused on a single domain; Androutsopoulos et al. note that "models convey many types of information better than programs, but become unwieldy in scale far quicker." (Androutsopoulos et al. 2013) This challenge is directly correlated with the scale of system models, and presents a burden for large, industrial projects.

1.1.2. Model Traversal

OSATE, which is based on Eclipse, has an open, plug-in based architecture. It provides plug-in authors with support for accessing a model's elements and their relationships using the visitor pattern (Gamma et al. 1995). While well-known and powerful, the visitor pattern relies on the model's structure and most naturally supports system analyses based on hierarchical decomposition into subcomponents. This can prove challenging to use, in particular for the many analyses that require a data-flow based ordering of components in a system. These analyses typically re-implement similar reachability calculation metapatterns, which can be a time-consuming and error-prone task.

1.2. The OSATE Slicer

This paper details a new software tool: a plug-in for OSATE, which we term the *OSATE Slicer* (or, where context makes it clear, simply the Slicer). It automatically constructs a reachability graph, which can then be queried by other plug-ins to support both manual and automated analyses or—in future work—used to create submodels specific to particular system aspects. Specifically, this paper makes two contributions:

1. **Description of OSATE Slicer:** We provide details on the reachability graph's definition, construction from an AADL model, and supported queries.

2. **Evaluation:** We evaluate the suitability of the queries for various assessments of system architectures, including both existing and potential analyses. We also compare the slicer's performance to existing state-of-the-art, finding significant performance improvements.

We present an overview of the technologies we build on in the next section. In Section 3 we discuss the Slicer, then review the state of the art in Section 4. We evaluate the Slicer's performance against this state of the art and its suitability for various applications in Section 5. We discuss future work in Section 6 and conclude in Section 7.

2. Technical Background

2.1. Slicing

Program slicing, as originally described, decomposes a program by analyzing its data and control flow to produce a minimal version of the program that still produces some desired subset of the original program's behavior (Weiser 1984). Other slicers, like the tool described in this paper, "highlight the identified subprogram within the larger program" (Androutsopoulos et al. 2013) but do not do any rewriting / do not produce a minimal program. Regardless of the output format, program slicing has become a well-established and broadly-applied technique with a number of refinements and specializations (Silva 2012).

2.1.1. Reachability Graphs

Directed graphs are a common formalization / data structure for slicing implementations, including in Weiser's original formalization and implementation (Weiser 1984). The specifics of these graphs differ in what the nodes and edges represent, but in general vertices represent some notion of program state and edges represent allowable, i.e., reachable, state transitions. Walks through—and projections over—the graph are often helpful, though their semantics (e.g., execution traces, impact of a single program statement) are dependent on the specifics of the vertex and edge definitions.

2.1.2. Model Slicing

Application of program slicing techniques to models is non-trivial task, for reasons both syntactic (e.g., graphical vs textual representations) and semantic (e.g., support for nondeterminism) (Androutsopoulos et al. 2013). The overlap between state-based model slicing and program slicing continues to be explored; notable examples include:

- **SafeSlice:** Operates on SysML, and its primary goal is requirement traceability rather than reachability analysis (Briand et al. 2014).
- **Ahmadi et al.'s Slicer:** Operates on UML-RT, and its primary goal is preservation of the structural and behavioral aspects of a reduced model (e.g., Weiser's original goal) (Ahmadi et al. 2018).
- **Kompren:** Operates on any metamodel and conformant model to automatically generate a slicer for the model (Blouin et al. 2015).
- **Awas:** Operates on AADL models, and is very closely related to this work (Thiagarajan et al. 2021). We use Awas as a basis for our evaluation; it is described in detail in Section 4.

2.2. MBE for Critical Systems: AADL, EMV2 and OSATE

Developers of critical systems who want to utilize model-based engineering have a number of options in terms of modeling languages and tools, including both OMG's System's Modeling Lanaguage (SysML) (*OMG Systems Modeling Language (OMG SysML), Version 1.6* 2019) and SAE International's Architecture Analysis and Design Language (AADL) (*SAE Architecture Analysis and Design Language (AADL)* 2022). For this work we used AADL because of its strong semantics which enable the single source of truth concept's realization, support for multi-domain modeling (including off-nominal behavior via its error

```
1  package BasicErrorFlow
2  public
3    system sys
4    end sys;
5    system implementation sys.impl
6      subcomponents
7        a : abstract Thing.one;
8        b : abstract Thing.two;
9        c : abstract Thing.three;
10     connections
11       aToB1 : feature a.o1 -> b.i1;
12       aToB2 : feature a.o2 -> b.i2;
13       aToB3 : feature a.o3 -> b.i3;
14       bToC1 : feature b.o1 -> c.i1;
15       bToC2 : feature b.o2 -> c.i2;
16       bToC3 : feature b.o3 -> c.i3;
17     annex EMV2 {** **};
18   end sys.impl;
19   abstract Thing
20     features
21       i1 : in feature;
22       i2 : in feature;
23       i3 : in feature;
24       o1 : out feature;
25       o2 : out feature;
26       o3 : out feature;
27   end Thing;
28   -- continued in Listing 2
```

Listing 1 A snippet of AADL's core language's textual syntax showing the BasicErrorFlow example.

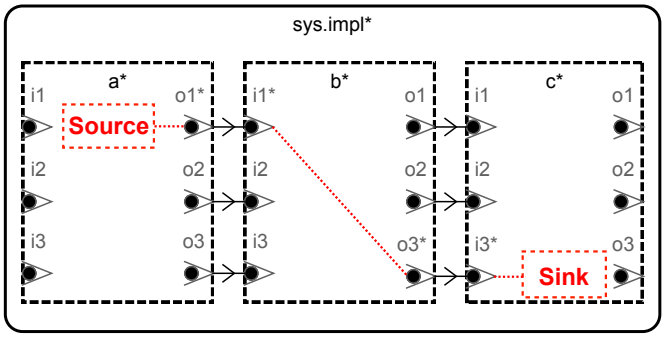

Figure 1 A snippet of graphical AADL, showing the BasicErrorFlow model from Listing 1. The black elements and lines are valid AADL graphical syntax; the red elements and lines are notional (as EMV2 does not have a graphical syntax) and correspond to the textual error-modeling annotations in Listing 2.

modelling language annex (*SAE Architecture Analysis and Design Language (AADL) Annex Volume 1* 2015)), and extensible analysis tooling that could benefit from slicing and reachability-based support.

2.2.1. Core Language: AADL

AADL is a language for modeling the architecture of critical, embedded systems. It supports the modeling of hardware (e.g., `processor`, `memory`) and software (e.g., `thread`, `subprogram`) elements, their interconnections and any access points or bindings between them. It also contains element types for aggregating other elements (`system`), placeholders for future modeling when details are either unnecessary or unavailable (`abstract`) and "black box" components where the interface is known but the decomposition and implementation details should be hidden (`device`).

AADL has both textual and graphical syntaxes; an example of its textual syntax is shown in Listing 1 and corresponding graphical syntax in Figure 1. In order to illustrate the concepts discussed in this section and the next, we will use a simple system architecture model called the *BasicErrorFlow* model. It consists of a single AADL `system`, which has a single `implementation` `sys.impl` and three `abstract` components: `a`, `b`, and `c`. Each component has three inputs (`i1`, `i2`, and `i3`) and three outputs (`o1`, `o2`, and `o3`).

Notable language features include the separation of an elements' interface from its implementation: observe that the `system` `sys` is declared on lines 3-4, but it has no ports or other means to communicate with its environment, so its interface is empty. However, the `abstract` component specification named "`thing`" has three input and three output `features` (essentially generic communication ports), which are declared in its interface on lines 19-27. `sys`'s implementation contains a decomposition specification: both subcomponents (lines 6-9) and their interconnections (lines 10-16) are specified as is an (empty, in this listing) extension via AADL's language annex mechanism (line 17).

Although AADL is a modeling language, its processing is similar to that of a programming language. Users specify a *declarative* model of a system: a collection of component specifications which tooling automatically compiles into a fully resolved abstract syntax tree called the *instance* model. This instantiation process is critical to resolve references, property values, etc.

2.2.2. Off-Nominal Language Annex: EMV2

In addition to the core elements of the language, AADL supports extensions into a number of related modeling domains via language *annexes*. A number of annexes exist for modeling e.g., data, nominal behavior, or off-nominal behavior, i.e., behavior in the presence of errors. We make extensive use of this latter annex. The Error Modeling Annex, (in its second revision, so it uses the abbreviation EMV2) has a number of features to support various safety and reliability-oriented system development and certification activities (*SAE Architecture Analysis and Design Language (AADL) Annex Volume 1* 2015). In this work, we relied heavily on EMV2's `error type`, `propagation`, and `error flow` mechanisms; Listing 2 shows a continuation of the previous listing with EMV2 annotations. These mechanisms broadly align with concepts from the *Fault Propagation and Transformation Calculus*, which was designed to support compositional safety and reliability analysis (Wallace 2005).

1. **Error Types:** One of the core features of EMV2 is the ability to define different error types, which are ways in which communication between system elements can be malformed, e.g., timing (a message arrives too early or too late) or value (the value of some data is incorrectly high or low). The EMV2 annex comes with a library of

```
29  -- continued from Listing 1
30  abstract implementation Thing.one
31    annex EMV2 {**
32      use types ErrorLibrary;
33      error propagations
34        o1 : out propagation {ItemTimingError};
35        flows
36          o1TimingSrc : error source o1 {
    ItemTimingError};
37      end propagations;
38    **};
39  end Thing.one;
40  abstract implementation Thing.two
41    annex EMV2 {**
42      use types ErrorLibrary;
43      error propagations
44        i1 : in propagation {ItemTimingError};
45        o3 : out propagation {ItemTimingError};
46        flows
47          i1ToO3 : error path i1 {ItemTimingError
    } -> o3 {ItemTimingError};
48      end propagations;
49    **};
50  end Thing.two;
51  abstract implementation Thing.three
52    annex EMV2 {**
53      use types ErrorLibrary;
54      error propagations
55        i3 : in propagation {ItemTimingError};
56        flows
57          i3TimingSink : error sink i3 {
    ItemTimingError};
58      end propagations;
59    **};
60  end Thing.three;
61 end BasicErrorFlow;
```

Listing 2 A snippet of textual AADL extending the BasicErrorFlow example from Listing 1 showing the error modeling (EMV2) language annex.

common error types, and is designed to be extended by users via domain- and model-specific type declarations (Procter & Feiler 2018). We use only one error type (`ItemTimingError`, see, e.g., line 34) in our example; it is part of the standard EMV2 Error Library which is referenced via the `use types` declaration on line 32.

2. **Propagation Paths:** The potential paths error tokens take between components are represented as propagation paths: these can be along connections, bindings (e.g., a bus fails, causing errors with messages sent on the connections that were bound to it), or any user-defined path through a system (which enables modeling propagations that do not follow a system's communication topology). Importantly, propagation paths are not part of the declarative model (i.e., they are created as part of the instantiation process). Rather, they are specified by the modeler in a piecemeal fashion as part of a component's interface, see e.g. line 45 of Listing 2 which specifies that the component `Thing.two` *may* propagate out an `ItemTimingError` on its `o3` feature.

3. **Error Flows:** The specification of how error tokens move through a component—i.e., are created, propagated, transformed, or consumed—is detailed by the component's error flows.

   (a) **Creation:** The instantiation of an error type at an `error source` creates a *token*; see line 36 of Listing 2.

   (b) **Propagation:** The transmission of an error token from one element to another. Used when an element cannot compensate for a particular error, e.g., if a sensor is connected to a bus and produces late output, the bus will *propagate* the late error tokens. The declaration on line 47 of Listing 2, specifies that an `ItemTimingError` token will be output on the component's `o3` output feature if a token of that same type arrives on the component's `i1` input feature.

   (c) **Transformation:** The conversion of a token of a particular type into a token of a different type. Used when a model element's behavior changes an error, but does not eliminate it, e.g., if a sensor is connected to a controller and produces late output, the controller may transform late error tokens into unsafe commands (not shown in Listing 2).

   (d) **Consumption:** The consumption of error tokens at an `error sink`; see line 57 of Listing 2.

2.2.3. Tooling: OSATE

The Open Source AADL Tool Environment (OSATE)[1] is the reference implementation for the AADL language. It supports editing and analyzing AADL models, including many of the more popular automated analyses across a number of domains including safety, security, performance, and others. It can be used directly by modelers, but has also been used by other tools as an instantiator / analyzer for automatically-generated AADL models. The OSATE Slicer also makes extensive use of OSATE's instantiator's recent extensions to support the EMV2 Annex. Much like the instantiation process for the core language, these extensions make EMV2 more amenable to automated analysis by, among other things, supporting easier model traversal and concretizing model elements whose specifications span multiple components or levels of the system hierarchy.

3. The OSATE Slicer

The OSATE Slicer calculates the reachability between different elements of an AADL model. It does this by generating, and then querying, two separate reachability graphs: one for *nominal* behavior, i.e., the connectivity and reachability between elements when the system is operating under expected conditions, and one for *off-nominal* behavior, i.e., connectivity and reachability between elements in the presence of errors. Both graphs are generated simultaneously from a single instance model, the nominal reachability graph from AADL's core elements and the off-nominal reachability graph from AADL's core elements as well as any available EMV2 annotations. In this section we first define the graph representation (Section 3.1), then discuss how the graph is generated from an input model (Section 3.2), and available traversals and queries are presented in Section 3.3.

[1] https://osate.org

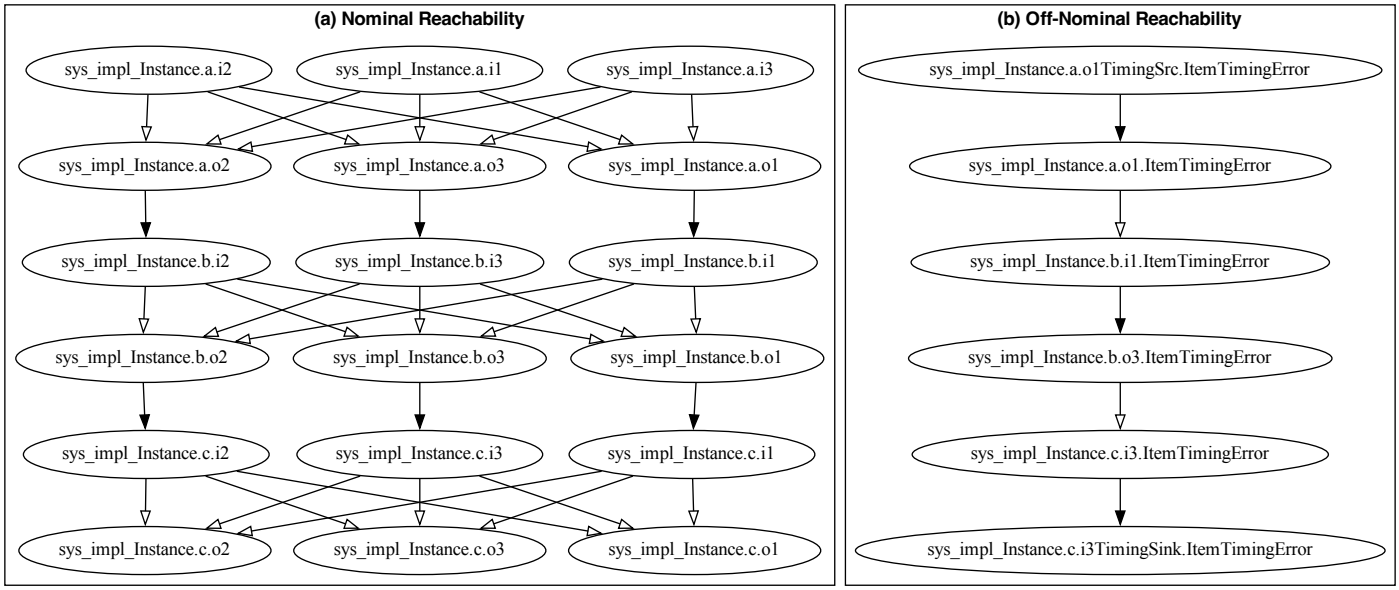

Figure 2 (a) Left, the nominal reachability graph for the BasicErrorFlow example. (b) Right, the model's off-nominal reachability graph. Filled arrowheads represent edges added in the first phase of the respective graph generation algorithms, edges with empty arrowheads are added in their second phase.

3.1. Graph Representation

3.1.1. Nominal Graph Representation The nominal graph representation $\mathcal{G}_N$ is comprised of a set of vertices and directed edge relation: $\mathcal{G}_N = (V_N, \rightarrow_{e_N})$:

1. Vertices (V_N) include all of a model's features (F) and the access points (A) used by connections: $V_N = F \cup A_{used}$
2. Edges ($\rightarrow_{e_N}$) are pairs of features that are directly (i.e., intransitively) connected: $\rightarrow_{e_N} \subseteq V_N \times V_N$

In the BasicErrorFlow example's nominal reachability graph, shown in Figure 2(a), there are 18 vertices, identified by their full path from the root of the system instance, i.e., $V_N = \{\text{sys\_impl\_Instance.}[a-c].[i,o][1-3]\}$. Edges are anonymous, but since only one edge is allowed per pair of vertices in our representation[2], they can be uniquely identified by their source and destination vertex's names.

3.1.2. Off-Nominal Graph Representation The off-nominal graph representation $\mathcal{G}_O$ is similarly comprised of a set of vertices and directed edge relation: $\mathcal{G}_O = (V_O, \rightarrow_{e_O})$. However, its vertex definition is extended to also include error information.

1. Vertices (V_O) are the elements of the model that can propagate error tokens (Propagations ($P_F \cup P_B \cup P_A \cup P_P$), Error Sources ($R_{Src}$), Error Sinks ($R_{Snk}$)) and the token ($T$) that is propagated, i.e., $V_O = L \times T$, $L = P_F \cup P_B \cup P_A \cup P_P \cup R_{Src} \cup R_{Snk}$:
   (a) Feature Propagations, P_F
   (b) Binding Propagations, P_B
   (c) Access Propagations, P_A
   (d) Point Propagations, P_P
   (e) Error Sources, R_{Src}
   (f) Error Sinks, R_{Snk}
   (g) Error Tokens, T
2. Edges are connections between vertices $\rightarrow_{e_O} \subseteq V_O \times V_O$

In the BasicErrorFlow's off-nominal reachability graph, shown in Figure 2(b), there are six vertices labeled by their full paths including the name of the propagated error token. So, $P_F = \{\text{sys\_impl\_Instance.}[\text{a.o1, b.i1, b.o3, c.i3}].\text{ItemTimingError}\}$, $R_{Src} = \{\text{sys\_impl\_Instance.a.o1TimingSrc.ItemTimingError}\}$, $R_{Snk} = \{\text{sys\_impl\_Instance.c.o3TimingSrc.ItemTimingError}\}$, $T = \{\text{ItemTimingError}\}$, $P_B = P_A = P_P = \emptyset$.

3.2. Graph Generation

The OSATE Slicer's implementation is open-source and publicly available[3]. Its reachability graphs are implemented using JGraphT (Michail et al. 2020). Edges use JGraphT's default representation, and are directed, anonymous, and unweighted. Vertices point to the AADL model elements (e.g., features, error sources / sinks) they are associated with; maintaining these pointers enables users of the Slicer to query linked AADL elements directly to, e.g., fetch associated property values (such as latency) or perform more in-depth processing. Generation of both the nominal and off-nominal graphs involves two distinct

[2] This is a restriction of our formalization (not AADL) and we can remove it if necessary. However, it simplifies the graph representation and does not impact the expressiveness of our query types.

[3] https://github.com/osate/osate2/tree/master/core/org.osate.slicer

Algorithm 1 Nominal Graph Generation. VERT($Elem$) and EDGE(Src, Dst) create (if necessary) and retrieve vertices and edges associated with supplied parameters. CONTAINER retrieves the model element that contains the supplied parameter.

Require: AADL instance Model m
Ensure: Populated set of vertices V_N and edges $\rightarrow_{e_N}$

1: **function** CONSTRUCTNOMINAL(m)
2: $ED := \emptyset$ ▷ Components with explicit decompositions
3: $\mathcal{G}_N = (V_N, \rightarrow_{e_N}) := (\emptyset, \emptyset)$ ▷ Initialize $\mathcal{G}_N$
4: **for** $feat \in m$ **do** ▷ $feat$ is a `feature`
5: $V_N := V_N \cup$ VERT($feat$)
6: **for** $conn \in m$ **do** ▷ $conn$ is a `connection`
7: $V_N := V_N \cup$ VERT($conn_{Src}$) $\cup$ VERT($conn_{Dst}$)
8: $\rightarrow_{e_N} := \rightarrow_{e_N} \cup$EDGE($conn_{Src}, conn_{Dst}$)
9: $ED := ED \cup$ CONTAINER($conn$)
10: **for** $flow \in m$ **do** ▷ $flow$ is a `end to end flow`
11: $\rightarrow_{e_N} := \rightarrow_{e_N} \cup$EDGE($flow_{Src}, flow_{Dst}$)
12: $ED := ED \cup$ CONTAINER($flow$)
13: **for** $comp \in (m \setminus ED)$ **do** ▷ $comp$ is a `component`
14: **for** $feat_{in} \in comp$ **do**
15: **for** $feat_{out} \in comp$ **do**
16: $\rightarrow_{e_N} := \rightarrow_{e_N} \cup$EDGE($feat_{in}, feat_{out}$)

Algorithm 2 Off-Nominal Graph Generation. VERT($Elem, T$) and EDGE(Src, Dst) create (if necessary) and retrieve vertices V_{Elem} and edges V_{Src}, V_{Dst} associated with supplied parameters.

Require: AADL instance model m
Ensure: Populated set of vertices V_O, partially populated set of edges $\rightarrow_{e_O}$, set of possible propagations PP

1: **function** CONSTRUCTOFFNOMINAL(m)
2: $PP := \emptyset$ ▷ Set of possible propagations
3: $\mathcal{G}_O = (V_O, \rightarrow_{e_O}) := (\emptyset, \emptyset)$ ▷ Initialize $\mathcal{G}_O$
4: **for** $(src, P_{src}, T_{src}) \in m_{R_{Src}}$ **do** ▷ R_{src} is a `error source`
5: $V_O := V_O \cup$ VERT(src, T_{src}) $\cup$ VERT(P_{src}, T_{src})
6: $\rightarrow_{e_O} := \rightarrow_{e_O} \cup$EDGE($V_{src}, V_{P_{src}}$)
7: **for** $(snk, P_{snk}, T_{snk}) \in m_{R_{Snk}}$ **do** ▷ R_{snk} is a `error sink`
8: $V_O := V_O \cup$ VERT(snk, T_{snk}) $\cup$ VERT(P_{snk}, T_{snk})
9: $\rightarrow_{e_O} := \rightarrow_{e_O} \cup$EDGE($V_{P_{snk}}, V_{snk}$)
10: **for** $ErrPath \in m$ **do** ▷ $ErrPath$ is a `error path`
11: $V_O := V_O \cup$ VERT($ErrPath_{dst}, T_{dst}$)
12: **for** $(src, T_{src}) \in ErrPath$ **do**
13: $V_O := V_O \cup$ VERT($ErrPath_{src}, T_{src}$)
14: $\rightarrow_{e_O} := \rightarrow_{e_O} \cup$EDGE($V_{ErrPath_{src}}, V_{ErrPath_{snk}}$)
15: **for** $PPath \in m$ **do** ▷ $PPath$ is a `propagation path`
16: $PP := PP \cup$ PPROP($PPath_{src}, PPath_{dst}$)

phases which we detail in the following sections; at a high-level the first phase uses the model's explicit declarations for generating most of the vertices and edges, the second phase generates additional vertices and edges based on implicit relationships derived from the semantics of AADL and EMV2.

3.2.1. Nominal Graph Generation **Phase 1** Generation of the nominal graph is fairly straightforward, as shown in Algorithm 1. First, a vertex is created for each of the model's `features` (lines 4-5) and an edge is created for every `connection` (lines 6-8). `connections` can also connect model elements which are not `features`; in this case only necessary vertices will be created, i.e., no vertex will be created for unused access points. Edges are also created if a flow specification instance describes an intra-component connection (lines 10-12). Edges added in this phase have filled arrowheads in Figure 2(a).

Phase 2 If a component is a leaf (i.e., it does not specify any `subcomponents` and their interconnections) and does not define relationships between its inputs and outputs using at least one `end to end flow`, we assume the worst case and map to every input feature to every output feature (lines 13-16). In Figure 2(a), these edges have empty arrowheads. While this strategy ensures that reachability queries do not produce false negatives, it can lead to a potentially significant number of false positives, i.e., multiple non-existent paths through the system may be identified. To reduce the number of non-existent paths, `flow` specifications should be provided by the modeler when possible.

3.2.2. Off-Nominal Graph Generation This is notionally similar to the previous case; however the specific sources of the vertices and edges differ, as does the strategy for generating intracomponent flows in the second phase.

Phase 1 The first phase, shown in Algorithm 2, begins by creating a vertex for each error source, a second vertex for the associated propagation (i.e., feature, binding, access, or point propagation) and then creating an edge from the source to the propagation (lines 4-6). Incoming propagations are mapped to sinks in a similar fashion (lines 7-9).

Because it is possible to have multiple errors transform into a single error without requiring nondeterminism (e.g., both a timing error and a value error could cause a component to fail), EMV2's `error path`s can have multiple error tokens associated with their source. However, the reverse (e.g., a timing error that could cause either a value error or component failure) requires nondeterminism and is thus not allowed: destinations can only have a single error token. Thus, each `error path` requires the creation of a single vertex for its destination, but potentially multiple for its source (one for each `error type`), as well as an edge from each source vertex to the destination vertex (lines 10-14). Edges added in this phase are represented with filled arrowheads in Figure 2(b).

Finally, we iterate over the model's `propagation path`s, which document potential links between one model element's outgoing `propagation` and another model element's incoming one. Importantly, these may not be feasible: if the required inputs cannot arrive at the source `propagation`, the propagation will not trigger and the outputs will not be propagated to the destination. It is the responsibility of the reachability analysis to create edges only for feasible paths. In order to support this

Algorithm 3 Fixpoint Calculation. POP removes an item from the stack, EDGE(Src, Dst) creates and retrieves edges associated with supplied parameters.

Require: AADL model's error sources R_{Src}, populated set of vertices V_O, partially populated set of edges $\rightarrow_{e_O}$, set of possible propagations PP
Ensure: Fully populated set of edges $\rightarrow_{e_O}$

```
1:  function CalculateFixpoint(R_Src, V_O, →_eO, PP)
2:      repeat
3:          →'_eO := →_eO
4:          for src ∈ R_Src do                    ▷ src is a error source
5:              edges := V_srcOut                 ▷ Outgoing edges of V_src
6:              while |edges| > 0 do
7:                  CurrEdge := Pop(edges)
8:                  src := V_CurrEdge_Dst         ▷ CurrEdge's dest.
9:                  for OutEdge ∈ src_Out do
10:                     edges := edges ∪ OutEdge
11:                 for Prop ∈ {PP|Src = src} do
12:                     tgt := Prop_Dst
13:                     NewEdge := Edge(src, tgt)
14:                     if NewEdge ∉ →_eO then
15:                         edges := edges ∪ NewEdge
16:                         →_eO := →_eO ∪ NewEdge
17:     until →'_eO = →_eO                        ▷ Halt when edge set is unmodified
```

calculation, performed in Phase 2, we collect source and destination `propagations` into a set of possible propagations for further analysis (lines 15-16).

Phase 2 The second phase, shown in Algorithm 3, starts at each `error source` and follows all possible edges, including those described in the set of possible propagations identified previously. If a possible propagation is feasible, it is concretized into an edge and the algorithm repeats in case this new edge opened up previously infeasible propagation paths; it terminates when propagating a token from each source no longer adds any edges to the graph. Edges added in this phase are represented with empty arrowheads in Figure 2(b).

We begin by selecting an `error source` (line 4) and initializing a stack of edges to the error source's outgoing edges (line 5). Then, as long as the stack isn't empty (line 6), an edge is popped from the stack, and we store its destination (the start of the next step, so we term it *src*). Next, *src*'s outgoing edges are pushed onto the stack (lines 7-10). We then get all possible propagations which have *src* as a source (line 11), get the target for each one and create an edge from *src* to the target (lines 12-13). If that edge is new to the graph, we add it to both the graph and the stack (lines 14-16); this modifies the graph which means CALCULATEFIXPOINT will repeat.

3.3. Graph Queries

Once the reachability graph has been created, we can perform various queries that may be of interest to users and automated analyses. These four query types align directly with the "query concepts"[4] identified by Thiagarajan et al., though we modify some of their nomenclature slightly (Thiagarajan et al. 2021).

Algorithm 4 Reachability Calculation, which is a simple breadth-first walk of the graph to construct the reachable subgraph. EDGE(Src, Dst) creates and retrieves the edge associated with supplied parameters, BFITER retrieves a breadth-first graph iterator for the supplied graph starting at the supplied vertex, SUBGRAPH creates an initially empty subgraph of the supplied graph.

Require: Graph $\mathcal{G}$, Slicing criterion v_{origin}
Ensure: Reachable subgraph $\mathcal{G}_{sub}$

```
1: function Reach(G, v_origin)
2:     G_sub = (V_sub, →_esub) := (∅, ∅) = SubGraph(G)
3:     V_sub := V_sub ∪ v_origin
4:     v_previous := v_origin
5:     for v_current ∈ BFIter(G, v_origin) do
6:         V_sub := V_sub ∪ v_current
7:         →_esub := →_esub ∪ Edge(v_previous, v_current)
8:         v_previous := v_current
```

3.3.1. Forward Reachability

The first query type, shown graphically in Figure 3(a) and termed *forward reachability*, answers the question "What model elements can this model element affect?" That is, given a slicing criterion of some model element e, this query returns a subgraph containing all vertices and edges that could be reached by data, events, or errors that leave e. Since an entire subgraph is generated, the results of this query may be quite large. The query is therefore likely to be most useful either to support visualization, or as a first step in a multi-stage query. The process for creating the subgraph, shown in Algorithm 4, is a straightforward breadth-first walk of the reachability graph, where visited vertices and edges are added to an initially-empty subgraph. Note that the process is transparent to callers and identical regardless of if the nominal or off-nominal graph is being queried: the nominal graph is used if the slicing criterion is a core AADL model element, the off-nominal graph is used if the criterion is an EMV2 element.

3.3.2. Backward Reachability

The second query type, shown graphically in Figure 3(b) and termed *backward reachability*, answers the question "What model elements can affect this model element?" That is, given a model element e, this query returns a subgraph containing all vertices and edges that can produce data, events, or errors that will reach e. Backward reachability is calculated with the same algorithm as forward reachability, the difference is that a reversed view of $\mathcal{G}$ (i.e., one with an identical vertex set but reversed edges) is queried.

3.3.3. Reachability From

The third query type, shown graphically in Figure 3(c) and termed *reachability from*, answers the question "Can this model element reach that one?"

[4] Thiagarajan et al. identify five query concepts; the fifth concept is queries involving errors. Since any of the Slicer's queries can use errors as a slicing criterion, we distinguish only four.

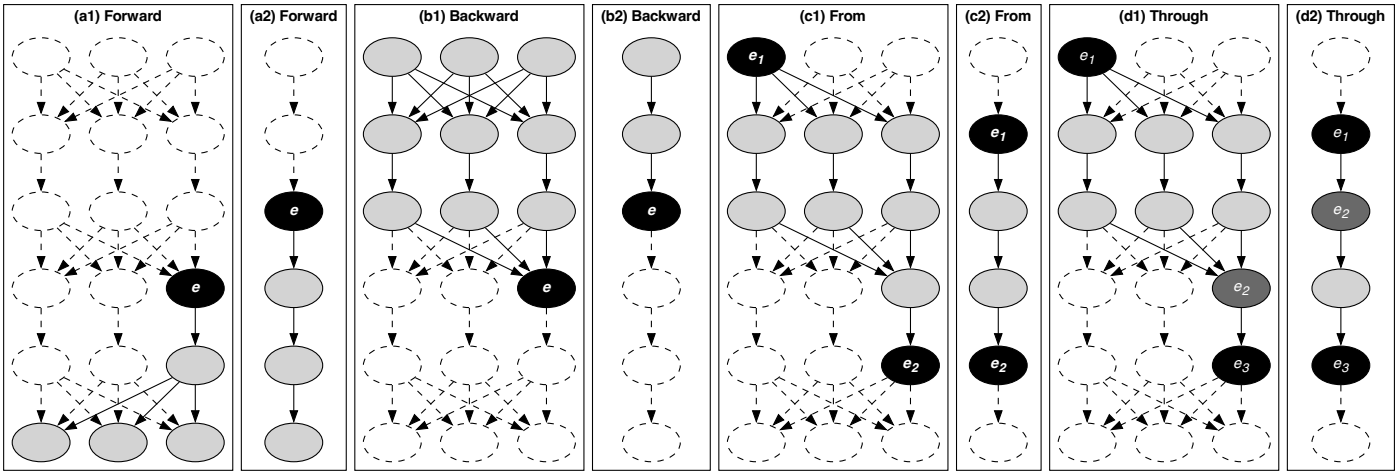

Figure 3 Visualization of various queries on the BasicErrorFlow reachability graphs from Figure 2. Origin and target (in parts (c) and (d)) vertices are shown in black, and labeled e. Vertices which will be part of the returned subgraphs (parts (a) and (b)) or are used in calculating the result (parts (c) and (d)) are grey; the special intermediate element used in reach through queries (d) is dark grey. Dashed edges and vertices are eliminated by the query.

Algorithm 5 Reachability Through Calculation, which verifies that all paths from an origin to a target pass through a specified midpoint, or finds a counterexample. $\text{PATH}(\mathcal{G}, v_{origin}, v_{target})$ finds a path from v_{origin} to v_{target} in $\mathcal{G}$, CUTPOINTS identifies the set of cutpoints in the supplied graph.

Require: Graph $\mathcal{G} = (V, \rightarrow_e)$, Vertices $v_{origin}, v_{mid}, v_{target}$
Ensure: Path $\mathcal{G}_{path}$ containing counterexample, or nothing

1: **function** REACHTHROUGH($\mathcal{G}, v_{origin}, v_{mid}, v_{target}$)
2: $\quad\mathcal{G}_{fwd} := \text{REACHFORWARD}(\mathcal{G}, v_{origin})$
3: $\quad\mathcal{G}_{back} := \text{REACHBACKWARD}(\mathcal{G}_{fwd}, v_{target})$
4: $\quad$**if** $v_{mid} \notin \text{CUTPOINTS}(\mathcal{G}_{back})$ **then**
$\quad\quad\triangleright$ Remove midpoint and all edges connected to it
5: $\quad\quad\mathcal{G}_{mask} := (V \setminus v_{mid}, \rightarrow_e \setminus (v_{mid}, \_\_) \cup (\_\_, v_{mid}))$
6: $\quad\quad\mathcal{G}_{path} := \text{PATH}(\mathcal{G}_{mask}, v_{origin}, v_{target})$
7: $\quad$**else**
8: $\quad\quad\mathcal{G}_{path} := (\emptyset, \emptyset)$

That is, given model elements e_1 and e_2, this query returns true if e_2 is reachable from e_1; this aligns with the slicing concept of *chopping* (Silva 2012). Since this query returns a boolean, the implementation is somewhat optimized in that it simply performs a forward reach from e_1 and then checks if e_2 is included in the returned subgraph's set of vertices.

As this query returns a boolean result rather than a subgraph, it is useful for automated analyses that pose safety- or security-specific questions such as "Can an error from this sensor be propagated into that actuator?" or "Can data from this classified source reach that untrusted component?"

3.3.4. Reachability Through

The fourth query type, shown graphically in Figure 3(d) and termed *reachability through*, answers the question "Do all paths from this model element which reach that one go through some intermediate element?" That is, given model elements e_1, e_2, and e_3, this query returns true if every path from e_1 to e_3 passes through e_2. In our implementation, we also support the retrieval of a counterexample if a path from e_1 to e_3 is found that doesn't use e_2.

This query type is essentially a more powerful (though slower) version of the previous type, and is designed to answer more subtle / realistic safety- or security-specific questions such as "Will errors from this sensor be mitigated by a filter before reaching that actuator?" or "Can data from this classified source reach that less-trusted component without first passing through an encryption module?"

The algorithm for this query type, shown in Algorithm 5, is less trivial than the previous three. First, we calculate the forward reachability from the origin (line 2), this gives the subgraph reachable from the origin. We then do a backwards reachability query on that subgraph from the target (line 3), this produces a subgraph with all paths from the origin to the target. We then calculate the set of cutpoints[5] of this second subgraph and see if it contains the specified midpoint. If not, we calculate a counterexample by creating a subgraph without the midpoint (line 5) and use it to find a path from the origin to the target (line 6). Note that our implementation contains a number of optimizations which are not shown in this pseudocode, e.g., checking for reachability immediately (in case the target is completely unreachable from the origin).

3.3.5. Other Queries

The previous four query types were the primary motivators in the Slicer's development, and will likely provide most of the utility from the reachability graph. There are, however, other interesting queries that are easily specified and / or implemented using reachability graphs, and

[5] *Cutpoints*, also called *cut* or *articulation vertices* are vertices whose removal will disconnect an otherwise connected graph (Chartrand et al. 2015, pg. 57).

we briefly discuss them here.

Model Validation Many analyses, both manual and automated, begin with the premise that an architectural model is well-formed according to some criteria, but establishing the validity of these assumptions can be challenging. As an example, a safety analysis might reasonably assume that all error sources produce error tokens that can reach error sinks, and all error sinks are reachable: if not, the model may be incomplete or incorrect. A new implementation of an existing safety analysis (the *Fault Impact Analysis*, described in more detail in Section 5.1.1) makes these assumptions and it was straightforward to implement verification of them using depth-first traversals of the off-nominal reachability graph.

Neighbors One of the strengths of AADL is its support for describing hierarchical (de)composition, which is flexible and powerful. This expressiveness is a boon to users but can prove challenging for tool developers.

One seemingly simple task is finding a component's *neighbors*, i.e., given some model element e, discovering what model elements produce e's input or consume its output. Finding these same-abstraction-depth neighbors can be challenging using only the instance model, however, since the AADL instantiation process "flattens" the hierarchy: if a process contains a thread that communicates with a thread in a separate process, extracting the relationship between the two processes is not straightforward. Our implementation's use of an element's full path through the system model as the vertex label, however, allows us to extract its abstraction depth (i.e., the number of elements it is contained within) as a derived attribute. With easy access to an element's abstraction depth and those that it communicates with—provided trivially by the incoming and outgoing edges of the element's associated vertex—neighbor calculation is also straightforward and in use by an existing safety analysis (the *Architecture Supported Audit Processor*, described in more detail in Section 5.1.1).

4. Related Work: Awas

Awas, an "AADL information flow and error propagation analysis framework" (Thiagarajan et al. 2021) is a software tool that addresses many of the same technical and technological challenges as the OSATE Slicer. In this section, we describe Awas to give a context for the evaluation of the Slicer; for a broader survey of related work the interested reader is referred to Thiagarajan's work or Androutsopoulos's survey of slicing on behavioral models (Androutsopoulos et al. 2013).

The conceptual differences between Awas and the Slicer center around different design goals: the Slicer aims to be smaller and more performant, while Awas is more flexible. To this end, compared to Awas, the Slicer has:

- **A Simpler Formalization:** The Slicer's formalization is relatively straightforward, and does not include first-class support for some AADL features (e.g., a component's abstraction hierarchy) that Awas does.
- **A Simpler Implementation:** This simpler formalization and use of JGraphT lends itself to a simpler implementation as well. Awas is built on top of Sireum[6], which is a "high-assurance system engineering platform" that supports a range of activities across the development lifecycle for critical, embedded systems (Hatcliff et al. 2018). Sireum makes use of AADL in some of these activities, but its scope extends beyond the architectural modeling that AADL supports. Sireum's use can pose some challenges if its full set of capabilities are not needed: it is implemented in a different language (Scala, though Java interaction is supported via a facade), a somewhat heavyweight installation (distribution via two .jar files totalling nearly 50MB with another 50MB of libraries, as opposed to the Slicer's <30KB .jar with 1.3MB of libraries), and a more complicated installation process. The benefits of this integrated approach are significant and will likely outweigh the costs for some use cases (see, e.g., (Cofer et al. 2022)).
- **A Stronger Adherence to OSATE Meta-Model:** The Slicer builds its reachability graph directly in-memory from the OSATE-supplied instance model, as opposed to Awas / Sireum's custom intermediate representation. Maintenance, performance, and ease-of-comprehension by clients (i.e., plugin authors) should be aided by this consistency.
- **No Inbuilt Support for Visualization:** Awas can generate HTML5-based visualizations which are rendered in and and can be interactively explored with a web-browser. The reachability graph can be explored by clicking on ports or connections, queried using stored or hand-written queries, and multiple levels of the hierarchy displayed in different windows.
- **No Arbitrary Query Support:** While queries in the Slicer are performed by calling specific API methods, Awas has a query language which allows the execution of arbitrary queries by either other plugins or end users.

5. Evaluation

We now turn to an evaluation of the OSATE Slicer along two dimensions: first, its suitability for some of the more common types of analyses performed by modelers, and second, its performance on a corpus of open-source system models.

5.1. Suitability for Analysis

Many of the analyses for AADL models are compositional: components are modeled individually by a user, and then an analysis (possibly, but not necessarily, automated) considers their composition into an entire system or subsystem. Safety analyses, for example, seek to answer questions about the overall system using EMV2 specifications from individual components; OSATE's built-in latency analysis calculates the end-to-end time necessary to respond to some input or actuate based on a sensed value. Many of these system-level properties are really analyses not of the entire system but of paths through it, it is these analyses that the Slicer can aid. Other analyses will not benefit, however, such as analyses that aggregate properties across entire systems or subsystems rather than individual paths (to calculate,

[6] https://sireum.org/

e.g., power consumption or system cost) or those that are focused on component-level properties such as task scheduling that is local to a processor. In this section we discuss the Slicer's utility in three domains of analysis: safety, security, and system latency. We find that reimplementations of existing analyses are far simpler (as they do not require extracting data flow from a visitor-pattern traversal of the model's abstract syntax tree) and less error prone (as the algorithms have been previously tested for correctness) than the original implementations.

5.1.1. Safety

Theory System safety relies heavily on concepts defined on paths through the system. The semantics of error propagation in AADL's EMV2 is based in large part on the *Fault Propagation and Transformation Calculus* which supports answering system-level questions (akin to forward and backward reachability queries) like "What happens if this component produces erroneous output?" or "How might erroneous input be propagated into that component?" (Wallace 2005)

Fault Tree Analysis, a popular system-level hazard analysis technique, also supports the identification of cut sets, which are sets of "component failures and/or event combinations that can cause" undesired events to occur (Ericson II 2016). These cut sets can be conceptualized and reasoned about as paths through the off-nominal behavior of the system.

Usage We are currently using the OSATE Slicer in two safety analyses. The first, the *Architecture Supported Audit Processor* (ASAP), is a plugin for OSATE that presents safety-specific views of a system (Procter & Hugues 2022). ASAP, which was first implemented using Awas, relies heavily on both forward and backward slicing and the neighbor-finding algorithm discussed in Section 3.3.5. While heavily automated, ASAP provides support for manual safety analysis and this usage of the Slicer aligns well with our goal of reducing the cognitive burden associated with analyzing multi-domain models (see Section 1.1.1). The models used by ASAP have been small enough that the performance impacts of switching from Awas to the OSATE Slicer have been noticeable but not significant. The reduction in complexity of use, however, (by, e.g., not needing a full Sireum installation) has accelerated development and simplified distribution.

The second safety analysis that the Slicer has been used in is a (currently in-development and unreleased) re-implementation of the Fault Impact Analysis (FIA), a relatively simple hazard analysis (see, e.g., Larson et al.'s description of the existing implementation (Larson et al. 2013)). The FIA produces a spreadsheet documenting, for each error source, the path of generated errors through a system, i.e., which components propagate or transform which errors. A straightforward application of the forward reachability query, the new implementation using the Slicer is far simpler and has fewer limitations than the existing version. For example, the existing implementation has a hardcoded limit on the length of the propagation path an error can take through a modeled system.

5.1.2. Security

Theory Much of the effort spent on securing critical systems focuses on *security by design*: establishing that various properties hold across the entire system, as opposed to preventing specific attacks. This higher-level focus can involve showing that the system's architecture guarantees various types of properties; two of the most common are *confidentiality* and *integrity* of information (Anderson 2020). Confidentiality—the need to keep secret information secret—can be at least partially established using data flow / reachability information: if secret data can be shown to only flow through specific components, then the whole system need not be scrutinized as closely as those specific components. Similarly, integrity—that information is genuine as opposed to forged—is easier to establish when it can be shown that only certain trusted components have had access to it.

The flow of malicious data and control through a system are also of interest to system modelers; recent work has also demonstrated the viability of AADL for the generation of attack trees, e.g., TAMSAT (Wortman & Chandy 2022). Attack trees are similar to fault trees except they focus on malicious activities rather than component failures or unintended occurrences.

Usage None of the existing AADL / OSATE tooling uses the Slicer for security analysis. However, there are two analyses where its usage is being evaluated. The first involves verification of a number of security properties specified by the Bell-LaPadula (BLP) security policy (Anderson 2020). Greenhouse et al. identify four classes of properties that are necessary to establish an AADL model's compliance with a BLP policy (Greenhouse et al. 2021); verification of three[7] of them could potentially be simplified with the OSATE Slicer:

1. **The Star Property:** This property specifies that high-security information should not be readable by a component that can send it to a low-security component. This can be established by examining the reachability of data leaving high-security ports.

2. **Architectural Consistency:** Greenhouse et al. describe a number of checks required to make sure that a model's "security attributes are consistent with AADL-specific structures" such as flow path specifications. This can be established by retrieving the path through the system described by the flow.

3. **Information Sanitization:** Violations of the star property are necessary in practice, e.g., high-security information is routinely transmitted through low-security components after it is encrypted. The challenge lies in making sure that it will always be encrypted; this can be established using the Reach Through query type described in Section 3.3.4.

The second security analysis where we expect the OSATE Slicer to prove helpful is in the generation of attack trees. Existing tooling for calculating attack trees (Delange et al. 2016) has proven difficult to maintain and less robust than desired;

[7] The fourth property identified by Greenhouse et al., the Simple Security Property, is verifiable locally and does not require system-level reasoning.

this stems in part from the difficulty of calculating reachability directly on the AADL instance model. This corresponds directly to our goal of simplifying model traversal (see Section 1.1.2).

5.1.3. System Latency

OSATE's latency analysis, which calculates the total latency along `end to end flow`s through a system, is one of the more popular and frequently-used system analyses (Feiler et al. 2009). Intuitively, the analysis is a candidate for re-implementation using the OSATE Slicer: the analysis calculates the sum of individual latencies along a path through a system, e.g., the processing times of components, the transmission delay along connections, or the delay associated with the type of sampling used by a particular sensor. What's more, the existing implementation is somewhat dated and complex; refactoring could service considerable technical debt. However, there are sophisticated features of the existing implementation, such as use of AADL's `modes`, which the Slicer does not currently support. Addressing these limitations will be necessary to reach feature parity, and likely enable broader use of the Slicer.

5.2. Performance

Our overall goals of increasing the usability of both manual and automated model analyses by improving tooling requires examining the user experience. Software tooling that is responsive and performant contributes to that goal: both because the usability of software is negatively impacted by poor performance and because some models may simply not be analyzable by unoptimized tooling. To that end, in this section we examine the execution times of various queries using the Slicer, and as a point of comparison, the execution times of the same queries using Awas.

5.2.1. Experimental Setup

All but one of the models in our testing corpus (the "Large" version of the Wheel Brake System) were originally used in the performance evaluation of Awas (Thiagarajan et al. 2021). We have made the models, our automated testing infrastructure, the exact queries executed, human-readable versions of the reachability graphs generated by both the OSATE Slicer and Awas, raw runtime and complexity data, and our notes on modifications required by the models (e.g., fixes for compilation or parse errors) available as part of the artifacts supporting this paper and online[8]. Tests were run on a 2019 MacBook Pro with a 2.4GHz Core i9 processor and 32GB of RAM.

As both graph generation and query calculation encountered significant speedups in subsequent executions (presumably due to the effects of caching and the JVM's just-in-time compilation), we ran the full evaluation suite twice for both the Slicer and Awas. We used the timing values from the second executions, and performed three distinct queries for each query type; the first query's execution time is shown in the "Cold Start" column of Tables 1 and 2 and the third query's execution time is shown in the "Hot Start" column. Minor differences in execution time are largely attributable to the different queries themselves: the slicing criteria / query parameters were selected somewhat arbitrarily, e.g., some are valid, some invalid, some return large subgraphs or involve significant portions of the graph while others are restricted to smaller subgraphs or can trigger short-circuit optimizations that one or both of the Slicer and Awas support. As the typical use case of the Slicer has not been well-established, it is not immediately clear whether the cold- or hot-start times represent a more accurate execution time for the average user. While it is unlikely that a user would rapidly and repeatedly generate or query the reachability graph, it is possible or even likely that some automated analyses will. For completeness, we have included both in our results.

[8] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=974364

5.2.2. Discussion

Though execution time varied depending on model, query type, and slicing criterion, in general we observed that the Slicer generated its graph and executed queries one to two orders of magnitude faster than Awas. Additionally, the OSATE Slicer successfully analyzed several queries that did not terminate (given a maximum execution time of 30 seconds) when using Awas.

We found the Slicer's performance to be satisfactory. Execution times are low compared to: Awas, the duration considered instantaneous by users (below 100-200ms (Seow 2008)), and other system modeling activities. Regarding the latter, we note that the Slicer's graph generation took between roughly 1-6% of the time required to instantiate the models, which is required to perform essentially any useful system analysis.

Differences in Formalization Strategy As discussed in Section 4, some of the performance difference is attributable to the formalization strategies employed by Awas and the Slicer, namely, that the OSATE Slicer's formalization (see Section 3.1) is relatively lightweight compared to that used by Awas. For example, Awas's formalization includes first-class support for the containment hierarchy specified by an AADL model: a component's associated vertex in the graph representation contains a subgraph of the component's decomposition; the OSATE Slicer does not store the hierarchical relationships between model elements directly, but can regenerate it due to its implementation strategy (see the discussion of finding a model element's neighbors in Section 3.3.5). Additionally, Awas's edge specification contains two binary relations as opposed to the single relation used by the Slicer; which the analysis uses depends on the granularity of the query. The primary impact of these differences are in the complexity of the implementation: the algorithms used by Awas for calculating query results are, in general, more complex than their Slicer counterparts.

6. Future Work

In our work on the OSATE Slicer, we have identified two primary avenues for future work: enhancing the design of the API used to query the reachability graph and moving from calculating model slices to automated rewriting. The first depends on identifying and profiling real-world uses and users of the Slicer, while the second is more of an engineering (and, to a lesser degree, scientific) challenge.

OSATE Slicer	Cold Start Runtime (µs)				Hot Start Runtime (µs)			
	Generate	Reach	From	Through	Generate	Reach	From	Through
Aircraft System	36359	4873	150	2700	2296	60	68	68
Display Manager	4166	138	83	356	2486	157	46	63
Flight Guidance System	9234	89	44	252	2675	101	50	243
Flight Guidance TwoGPS	4523	176	57	235	3055	80	61	82
Isolette	955	117	45	182	733	36	35	142
Speed Regulation	9037	363	136	1008	10193	266	86	513
Simple UAV	2950	91	60	240	1896	79	33	203
AFRL UxAS	4291	388	239	312	3486	207	204	354
Wheel Brake System	3683	276	52	267	4123	51	35	154
Wheel Brake System (Large)	9627	601	443	664	8149	434	476	772

Table 1 Complexity and runtime data when using the OSATE Slicer on a collection of open-source AADL models. Vertices and edges refer to elements of the reachability graph generated by the OSATE Slicer. Cold Start runtimes refer to the first graph generation, forward reach, reach from, and reach through queries, Hot Start runtimes refer to the third.

Awas	Cold Start Runtime (µs)				Hot Start Runtime (µs)			
	Generate	Reach	From	Through	Generate	Reach	From	Through
Aircraft System	668252	74669	30738	4696	74550	1927	1607	2336
Display Manager	249359	11367	14463	47066	137806	4728	2610	2112
Flight Guidance System	165695	3187	2021	9767	14620	2382	2094	4600
Flight Guidance TwoGPS	74854	2101	2456	3681	17828	1474	4219	8025
Isolette	36083	1960	2578	11196	25408	1391	3448	3583
Speed Regulation	139226	2512	6878	91183	62313	4459	4133	92564
Simple UAV	82561	4104	2760	11899	46127	751	2028	769
AFRL UxAS	162572	4658	15023	—	141311	4137	15228	—
Wheel Brake System	131787	786	2091	3303	45967	539	907	16816
Wheel Brake System (Large)	1203040	27476	78998	88022	1058152	24480	81478	—

Table 2 Complexity and runtime data when using Awas on a collection of open-source AADL models. Vertices and edges refer to elements of the flow graph generated by Awas. Cold Start runtimes refer to the first graph generation, forward reach, reach from, and reach through queries, Hot Start runtimes refer to the third. — denotes that execution did not terminate within 30 seconds.

6.1. API Design

The design of the API has been, and will continue to be, informed by the needs of the analyses that use it. So far, this has only been a small number of safety analyses (as discussed in Section 5.1.1) and some early planning work for use in a security analysis (see Section 5.1.2). As the number of analyses and domains of application expand, we expect to broaden the range of queries and tune the specific parameters used by the various query types.

6.1.1. Additional Query Support The initial set of four reachability queries that the Slicer supports were aligned with common slicing concepts, the capabilities of Awas (Thiagarajan et al. 2021), and the capabilities necessary to support existing analyses. Additional queries (discussed in Section 3.3.5), came from applying the Slicer in previously unintended ways; we expect that other novel uses of the reachability graph will continue to present themselves as we expand its range of use. Androutsopolous et al.'s survey (Androutsopoulos et al. 2013) also presents a number of slicing types that we do not support (e.g., dynamic slicing (Guo & Roychoudhury 2008) or environment-based slicing (Androutsopoulos et al. 2011)) but that could potentially be quite useful given AADL's domains of application in embedded, critical systems.

6.1.2. Graph Exposure Whether or not to expose the raw reachability graph via the API, which would enable analysis authors to perform their own queries, is an ongoing discussion. While undoubtedly useful to sophisticated users, exposure of the graph comes with risks as well. These range from inflexibility for future structural changes and optimizations, plug-in authors inadvertently writing redundant or poorly-optimized queries, and the need to ensure that the underlying graph is not modifiable by incorrect or malicious plug-ins.

6.2. Model Rewriting

The Slicer produces slices of models by highlighting paths through them. As originally described, however, slicing involved actually re-writing programs to produce smaller programs that only had program statements that affected the slicing criterion (Weiser 1984). Supporting such re-writing for AADL models would potentially be useful for automated analyses, which could instantiate and operate on the smaller model, saving significant computational resources. Its utility for manual debugging, though, would likely be even higher: production of minimal examples helps with system comprehension and is a routine but challenging task in debugging, error reproduction, and other model-development tasks.

7. Conclusion

In this paper we have presented the design, implementation, and an evaluation of the OSATE Slicer. Our goals for the work were to ease existing challenges with model traversal (by automated analyses) and model understanding (by human users performing manual analyses). Though early, the tool makes progress on these goals, its applicability and utility across different analyses is promising, and its performance is better than existing state-of-the-art.

Acknowledgments

The author would like to thank Thiagarajan et al. for their assistance in accessing and analyzing the corpus of models used in Section 5.2. Additionally, Joe Seibel's work on the EMV2 instantiator and extensive supporting documentation was invaluable both in the creation of the Slicer and the text of this manuscript.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM23-0448.

References

Ahmadi, R., Posse, E., & Dingel, J. (2018, October). Slicing UML-based Models of Real-time Embedded Systems. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 346–356). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3239372.3239407

Anderson, R. (2020). *Security Engineering, 3rd Edition* (Third ed.). John Wiley & Sons, Inc.

Androutsopoulos, K., Binkley, D., Clark, D., Gold, N., Harman, M., Lano, K., & Li, Z. (2011, May). Model projection: simplifying models in response to restricting the environment. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 291–300). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1985793.1985834

Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., & Tratt, L. (2013, August). State-based model slicing: A survey. *ACM Computing Surveys*, *45*(4), 53:1–53:36. doi: 10.1145/2501654.2501667

Blouin, A., Combemale, B., Baudry, B., & Beaudoux, O. (2015, February). Kompren: modeling and generating model slicers.

Software & Systems Modeling, *14*(1), 321–337. doi: 10.1007/s10270-012-0300-x

Briand, L., Falessi, D., Nejati, S., Sabetzadeh, M., & Yue, T. (2014, February). Traceability and SysML design slices to support safety inspections: A controlled experiment. *ACM Transactions on Software Engineering and Methodology*, *23*(1), 1–43. (Publisher: ACM PUB27 New York, NY, USA) doi: 10.1145/2559978

Chartrand, G., Lesniak, L., & Zhang, P. (2015). *Graphs and digraphs* (Sixth edition. ed.). Boca Raton, FL: Chapman and Hall/CRC, an imprint of Taylor and Francis.

Cofer, D., Amundson, I., Babar, J., Hardin, D., Slind, K., Alexander, P., … Shackleton, J. (2022, May). Cyberassured Systems Engineering at Scale. *IEEE Security & Privacy*, *20*(3), 52–64. (Conference Name: IEEE Security & Privacy) doi: 10.1109/MSEC.2022.3151733

Delange, J., Nam, M.-Y., Feiler, P. H., & Klieber, W. (2016, January). An Architecture-Centric Process for MILS Development. In S. Tverdyshev (Ed.), . Prague, Czech Republic: Zenodo. doi: 10.5281/zenodo.47976

Ericson II, C. A. (2016). *Hazard Analysis Techniques for System Safety* (Second ed.). Fredericksburg, Virginia, United States of America: John Wiley & Sons, Inc.

Feiler, P., Hansson, J., de Niz, D., & Wrage, L. (2009). *System Architecture Virtual Integration: An Industrial Case Study* (Tech. Rep.). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns : elements of reusable object-oriented software* (37th printing. ed.). Reading, Mass: Addison-Wesley.

Greenhouse, A., Hansson, J., & Wrage, L. (2021, March). *Modeling and Validating Security and Confidentiality in System Architectures* (Tech. Rep. No. CMU/SEI-2021-TR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Guo, L., & Roychoudhury, A. (2008). Debugging Statecharts Via Model-Code Traceability. In T. Margaria & B. Steffen (Eds.), *Leveraging Applications of Formal Methods, Verification and Validation* (pp. 292–306). Berlin, Heidelberg: Springer. doi: 10.1007/978-3-540-88479-8_21

Hatcliff, J., Larson, B. R., Belt, J., Robby, & Zhang, Y. (2018). A Unified Approach for Modeling, Developing, and Assuring Critical Systems. In T. Margaria & B. Steffen (Eds.), *Leveraging Applications of Formal Methods, Verification and Validation. Modeling* (pp. 225–245). Cham: Springer International Publishing. doi: 10.1007/978-3-030-03418-4_14

Larson, B., Hatcliff, J., Fowler, K., & Delange, J. (2013, November). Illustrating the AADL Error Modeling Annex (v.2) Using a Simple Safety-Critical Medical Device. *ACM SIGAda Ada Letters*, *33*(3), 65–84. (Publisher: ACM ISBN: 978-1-4503-2467-0) doi: 10.1145/2658982.2527271

Michail, D., Kinable, J., Naveh, B., & Sichi, J. V. (2020, May). Jgrapht—a java library for graph data structures and algorithms. *ACM Trans. Math. Softw.*, *46*(2).

OMG Systems Modeling Language (OMG SysML), Version 1.6 (Tech. Rep.). (2019, November). Object Management Group.

Procter, S., & Feiler, P. (2018). The AADL Error Library : An Operationalized Taxonomy of System Errors. In *High Integrity Language Technology (HILT) Workshop.* Boston, MA.

Procter, S., & Hugues, J. (2022, June). Architecture-Supported Audit Processor: Interactive, Query-Driven Assurance. In *11th European Congress on Embedded Real-Time Systems (ERTS 2022).* Toulouse, France. Retrieved from https://hal.science/hal-03701277

SAE Architecture Analysis and Design Language (AADL) (Tech. Rep.). (2022, April). SAE International.

SAE Architecture Analysis and Design Language (AADL) Annex Volume 1 (Tech. Rep.). (2015, September). SAE International.

Seow, S. (2008). *Designing and engineering time: The psychology of time perception in software* (1st edition ed.). Addison-Wesley Professional.

Silva, J. (2012, June). A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, *44*(3), 12:1–12:41. doi: 10.1145/2187671.2187674

Thiagarajan, H., Hatcliff, J., & Robby. (2021, July). Awas: AADL information flow and error propagation analysis framework. *Innovations in Systems and Software Engineering*. doi: 10.1007/s11334-021-00410-w

Wallace, M. (2005). Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In *Proceedings of the Second International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA 2005)* (Vol. 141, pp. 53–71). doi: 10.1016/j.entcs.2005.02.051

Weiser, M. (1984, July). Program Slicing. *IEEE Transactions on Software Engineering*, *SE-10*(4), 352–357. (Conference Name: IEEE Transactions on Software Engineering) doi: 10.1109/TSE.1984.5010248

Wortman, P. A., & Chandy, J. A. (2022, December). A framework for evaluating security risk in system design. *Discover Internet of Things*, *2*(1), 7. doi: 10.1007/s43926-022-00027-w

About the author

Sam Procter is a Senior Researcher at the Software Engineering Institute at Carnegie Mellon University and leader of the Model Based Engineering Initiative. His research interests focus on tool support for system safety, particularly those with an architecture-centric approach. You can contact the author at sprocter@sei.cmu.edu or visit https://samprocter.com.