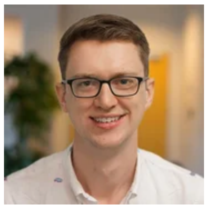
The OSATE Slicer: Fast Reachability Query Support for Architectural Models



SAM PROCTER

NOVEMBER 13, 2023

PUBLISHED IN

Software Architecture

CITE

https://doi.org/10.58012/wfgg-5t38

Get Citation

Systems whose failure is intolerable, often termed *critical systems*, must be designed carefully, regardless of whether they are safety-, security-, mission-, or life-critical—or some combination of the four. A range of development methodologies and technologies exists to support this careful design, but one of the more well-studied and promising is model-based engineering (MBE) where models of a system, subsystem, or a collection of components are built and analyzed. Due to the sophistication of these models and the intricacies of their analyses, however, software tooling is virtually required for all but the simplest tasks. In this post, I describe a new extension to the *Open Source AADL Tool Environment* (often abbreviated as OSATE), SEI's software toolset for MBE. This extension, called the OSATE Slicer, adapts a concept called *slicing* to architectural models of embedded, critical systems. It does this by calculating of various notions of reachability that can be used to support both manual and automated analyses of system models.

Before diving into the details, let me take a step back and discuss the process of model-based engineering in a bit more depth. Often, models are constructed and analyzed prior to the final construction of the component or system itself, leading to the early discovery of system integration issues. While engineering models are useful by themselves (e.g., communicating between stakeholders and identifying gaps in requirements) they can also be analyzed for various functional or non-functional system properties. What's more, if the model is built using a sufficiently rigorous language, these analyses can be automated. Models are, by definition, abstractions of the entities they represent, and those abstractions emphasize a particular perspective. But one thing that analyses—both manual and automated—can struggle with is interpreting a model built to showcase one perspective (e.g., a functional model of a system's architecture) from a different perspective (e.g., the flow of data or control sequences through those functional elements).

This particular shift in perspective is often necessary, though, and it underlies many of the manual and automated analyses we have created here on the MBE team at the SEI. Whether it's a safety analysis that needs to consider the flow of erroneous sensor readings through a system, a security analysis that must guarantee confidential data cannot leak out unencrypted ports, or a performance analysis that calculates end-to-end latency, the need to extract the paths that data or control messages take through a system is well established.

The OSATE Slicer

Recent work done by the MBE team aims to help calculate these paths through models of a system's architecture. We have created a software implementation that generates a graph-based representation of the paths through a system, and then uses that graph to answer *reachability queries*. This idea may sound familiar to some readers: it underlies the concept of program or model slicing, which is very closely related to our work, hence the software tool's name: *The OSATE Slicer* (or, where context makes it clear, just the slicer). The basic idea of slicing is to take a program or model and some input called a *slicing criterion*, and then discard everything that doesn't have to do with the slicing criterion to produce a reduced version of the program or model. While our work does not yet support this full vision of model reduction, the reachability graph and query support we have implemented are a necessary first step, and—as we discuss in this post—useful in their own right.

Like a lot of the work done by the SEI MBE team, this project was enabled by two key SEI technologies. First, the *Architectural Analysis and Design Language* (AADL) is an architecture modeling language for critical systems. It has well-specified semantics that make it particularly amenable to automated analyses, and has been used for decades by the U.S. Department of Defense (DoD), industry, and researchers for a variety of purposes. The second key technology is OSATE, which is an integrated development environment for AADL. Many analyses that operate on AADL models are implemented as plug-ins to OSATE, and the slicer is as well.

If you're not familiar with AADL, there are a number of resources available to explain the ins and outs of the language (the AADL website in particular is a great starting point). In this post, though, I'll use a simple model to illustrate some of the details of the OSATE Slicer. This model, shown below, is called the *BasicErrorFlow* example. It includes both core AADL, which specifies the basic architecture of a system, and annotations from AADL's EMV2 Language Annex, which extends the core language so that error behavior can also be modeled.

The black boxes and lines in the model below are valid AADL (which has both a graphical and a textual syntax) that show three communicating abstract (i.e., undefined and intended for later refinement) elements. Those elements communicate over features, named "i" for input or "o" for output, and numbered 1-3. Superimposed on top of this (in red) in a notional syntax is an example error flow from element a, through element b, into element c. You might imagine element a as some type of sensor that's prone to a particular failure, b as an automated controller which interprets that sensor data and issues commands based upon them, and c as some sort of actuator which effectuates the commands.

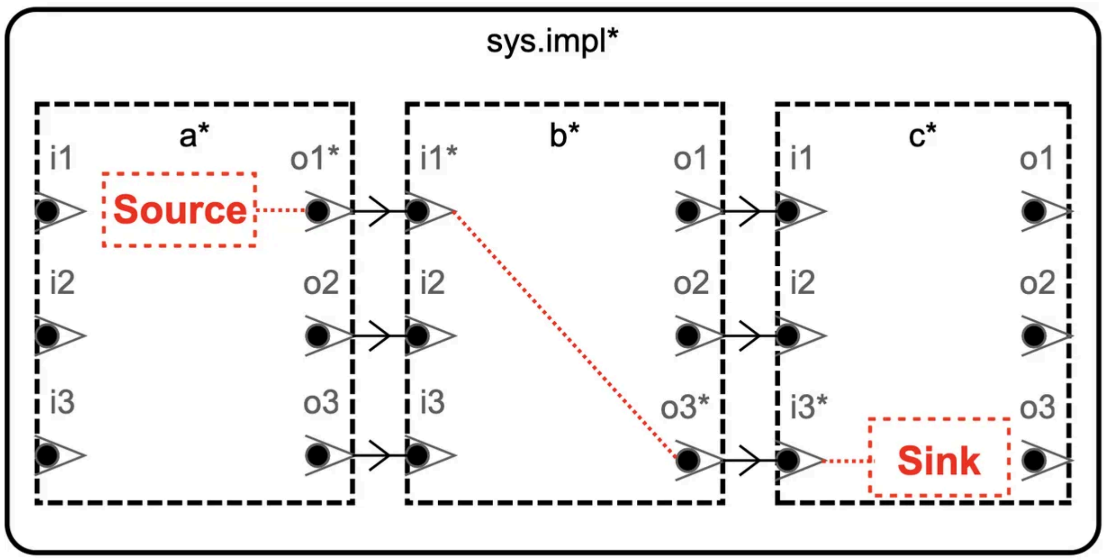


Figure 1: A snippet of graphical AADL, showing the BasicErrorFlow model

“Under the Hood” of Architectural Model Analysis

Let’s dive a bit deeper into how those analysis plug-ins typically work. Like many tools that process inputs specified in some sort of programming or modelling language, OSATE provides plug-in developers access to AADL model elements using a technique called the visitor pattern. Essentially, this pattern guarantees that every element will be “visited” and when it is, the developer of an analysis plug-in can specify some action to take (e.g., recording an associated property value or storing a reference to the element for later use). Significantly, though, the order in which those elements are visited has little to no bearing on the order in which they might create or access data or control messages when the system is operational. Instead, they are visited according to their position in the model’s abstract syntax tree.

Previous work done as part of the Awas project by Hariharan Thiagarajan and colleagues at Kansas State University’s SAnToS Lab in collaboration with the SEI demonstrated the value of extracting and querying a reachability graph from AADL models. That work was subsequently built on by projects both here at the SEI and externally. See, for example, its use in DARPA's *Cyber Assured Systems Engineering* (CASE) program. We were convinced of the value of this approach, but wanted to see if we could create our own implementation which—while simpler and less feature-rich than Awas—could be more well aligned with OSATE's implementation and design principles, and in doing so, could be more maintainable and performant.

Maintainability and Performance via Careful Design

Graph Definition and Implementation

Earlier in the post, I mentioned how the OSATE Slicer generates and queries something called a *reachability graph.* The term *graph* is used here to mean not a chart comparing different values of some variable, but rather a mathematical or data structure where *vertices* are linked together by *edges*, (i.e., "a collection of vertices and and edges that join pairs of vertices"). The reachability part of the term refers to the meaning of the graph: vertices represent particular elements of the system architecture, and if two vertices are connected by an edge, that signifies that data or control messages can flow from the model element associated with the source vertex to the element associated with the destination vertex. The simplest graph definition is just $G=(V,\rightarrow)$, and this is the definition we use: V is the set of architectural elements, and $\rightarrow$ is a function connecting some of those elements to some other elements. The devil is in the details, of course; in this case those details are *which elements* are included in V and *which relationships* are included in $\rightarrow$. These details are specified and explained in a paper published earlier this year on the work.

While our graph definition is simple, which should help achieve our goal of making it fast and straightforward to generate and query, it's still only a mathematical abstraction. We need to represent the graph in software, and for that we turned to the excellent and well-established graph theory library JGraphT. Encoding our graph in JGraphT was straightforward: we could associate OSATE's representation of AADL elements with JGraphT vertex objects, which lets analyses easily use both the graph and its associated system model. Practically, this means that analyses can run operations on the reachability graph, which will yield graph objects, such as subgraphs or individual vertices, and then translate those objects to AADL model elements that will be meaningful to users.

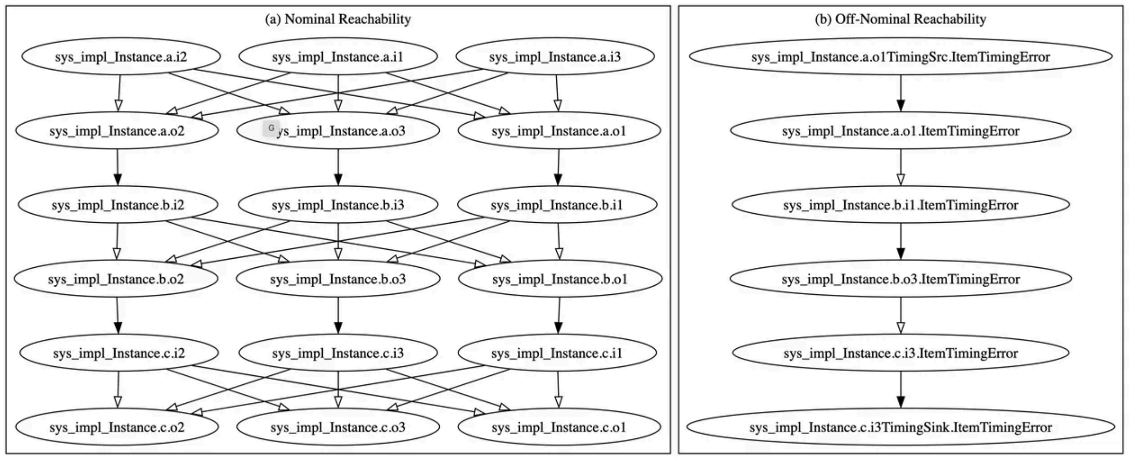


Figure 2: The reachability graph for the BasicErrorFlow model

The reachability graph for the BasicErrorFlow model introduced earlier is shown in Figure 2. There are a couple notable things about the graph: First, it’s actually two graphs, the one on the left is the *nominal* graph, constructed using only *core* AADL, which is the base language. The (far simpler) graph on the right is the *off-nominal* graph, constructed using both core AADL and its error-modeling extension known as EMV2. For the precise meanings of the graphs, I’ll again refer interested readers to the paper. For this post, I’ve included them to give an intuitive feeling of the sort of data structures we’re working with. The basic idea, though, is that a more detailed model produces a less ambiguous reachability graph; so the off-nominal graph (which can utilize the error flow information present in the model) is much simpler and more precise.

Querying the Reachability Graph

To get any value out of the reachability graph, we have to be able to *query* it, pose questions about relationships between various vertices. There are four foundational queries: reach *forward*, reach *backward*, reach *from*, and reach *through*.

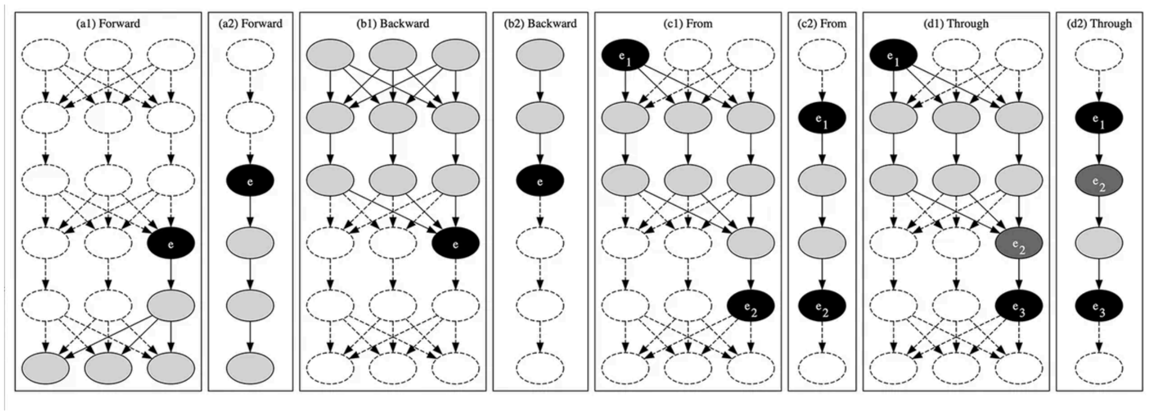


Figure 3: Queries of the reachability graph for the BasicErrorFlow model

Reach Forward and Backward

The first two queries are fairly straightforward. Reach forward queries ask, *What model elements can this model element affect?* That is, if we return to our conceptualization of the BasicErrorFlow model as a sensor connected to a controller connected to an actuator, we might ask, *Where can data readings produced by the sensor, or any commands derived from them, go?* Reach backward queries are similar, but they instead pose the question, *What model elements can affect this model element?* Applied to a real-world system, these queries might ask, *What sensors and controllers produce information used to govern this particular actuator?*

Figure 3 shows graphically, in (a1) and (a2), example forward reachability queries on the reachability graphs: nominal in (a1), off-nominal in (a2). Similarly, (b1) and (b2) show example backward reachability queries. The element used as the slicing criterion, i.e., the query origin, is shown in black and labeled with an *e*. The results of the query are all shaded elements—including the query origin. Notably, the result of executing this query is a reduced portion of a system's associated reachability graph (specifically an *induced subgraph*). Unlike some of the other queries that return a simple yes/no-style result, these subgraphs aren't likely to be very useful by themselves in automated analyses, and they don't lend themselves to, for example, DevOps-style automated evaluation. They are likely to be useful, though, for either generating visual results that can then be interpreted by a human, or as the first stage in more complex, multi-stage queries.

Reach From

The third query type is one of those multi-stage queries, though it’s not terribly complex. In reach from queries, we simply ask, *Can this model element reach that one?* We do this by first executing a forward reach query from the first element (*e1* in (c1) and (c2) in Figure 3) and then seeing if the second element (*e2*) is contained in the resulting subgraph. Knowing whether information from a sensor, or commands from a controller, can affect a particular actuator is useful, but this query really shines when executed on the off-nominal reachability graph. Recall that it is constructed using a system’s architecture (specified in AADL) and information about what happens when the system encounters errors (specified in the error-modeling extension to AADL called EMV2). This design means that reach from queries let modelers or automated analyses ask, *Can an error from this device reach that one, or is it somehow stopped?*

Reach Through

The fourth and final foundational query type answers questions of the form, *Do all paths from this model element which reach that one go through some particular intermediate element?*

The utility of this query may not be immediately obvious, but consider two scenarios. The first, from the safety domain, involves (a) a sensor that is known to occasionally produce jittery values, (b) a “checker” model element that can detect and discard those jittery readings, and (c) an actuator, which actuates in response to the sensor readings. We may want to check that all paths from the sensor (i.e., the origin, or *e1* in (d1) and (d2) in Figure 3) to the actuator (*e3*) go through the checker (*e2*)—hardly a simple task in a system where there may be multiple uses of the sensor’s data by a number of different intermediate controllersor other system elements.

In a second scenario from the domain of information security, some secret information must be sent across an untrusted network. To maintain secrecy, we should encrypt the data before broadcasting it. But how can we determine that there are no “leaks,” i.e., that no system elements processing or manipulating the secret information can send it directly or indirectly to the broadcasting element without its first passing through the encryption module? We can use the reach through query, with the source of the secret information being the origin, the encryption module being the intermediate element, and the broadcasting element the target.

Other Queries

From these four foundational queries, developers building automated analyses in OSATE can create more complex queries that ultimately can answer deep questions about a system. The utility of this approach is something we explored in our evaluation of the OSATE Slicer.

How Well Did We Do?

After creating the OSATE Slicer, we wanted to evaluate both how useful it is and how well it performs. In general, we were pleased with the results of our work, though as always, there's more to be done.

How Useful is the OSATE Slicer?

The first place we used the slicer was in the *Architecture Supported Audit Processor* (ASAP), an experimental automated safety analysis. ASAP had originally been created using Awas, but maintaining that dependency proved challenging. We were able to replace Awas with the Slicer in our implementation of ASAP. Doing so was relatively straightforward; while most of our existing implementation transferred seamlessly, we did have to write one custom query (described further in the paper).

The second place we used the OSATE Slicer is in an as yet unpublished re-implementation of OSATE's existing *Fault Impact Analysis* (described in, e.g., this paper by Larson et al.), which considers where a particular element's fault or error can go (i.e., be propagated to) in a fully-specified system. This was trivial to reimplement using the forward slice query, and then—as part of an ongoing research effort—we were able to take things a step further with a handful of custom queries to validate foundational assumptions about a system model that must be true for the analysis's results to be valid.

Looking forward, we've identified two potential security analyses that we are interested in updating to use the OSATE Slicer: an attack-tree calculator and a verifier that checks if a system meets the Bell-LaPadula security policy. Beyond that, there are other analyses that, at their core, explore properties of paths through a system. These can potentially benefit from the OSATE Slicer, though some are quite complex and may require additional features to be added to the Slicer.

How Fast is the OSATE Slicer?

In their publication on Awas, Thiagarajan et al. analyzed a corpus of 11 system models written in AADL. We set out to run the OSATE Slicer on this same corpus so that we could compare the performance of the two tools. Unfortunately, while many of the models were open-source, version information and other key specifics necessary for reproducibility are not present in their publication. We were able to work directly with them (we owe them thanks for that) as part of this effort to get access to most of those models and specifics, though, and have made an archive of the corpus available publicly as part of this effort.

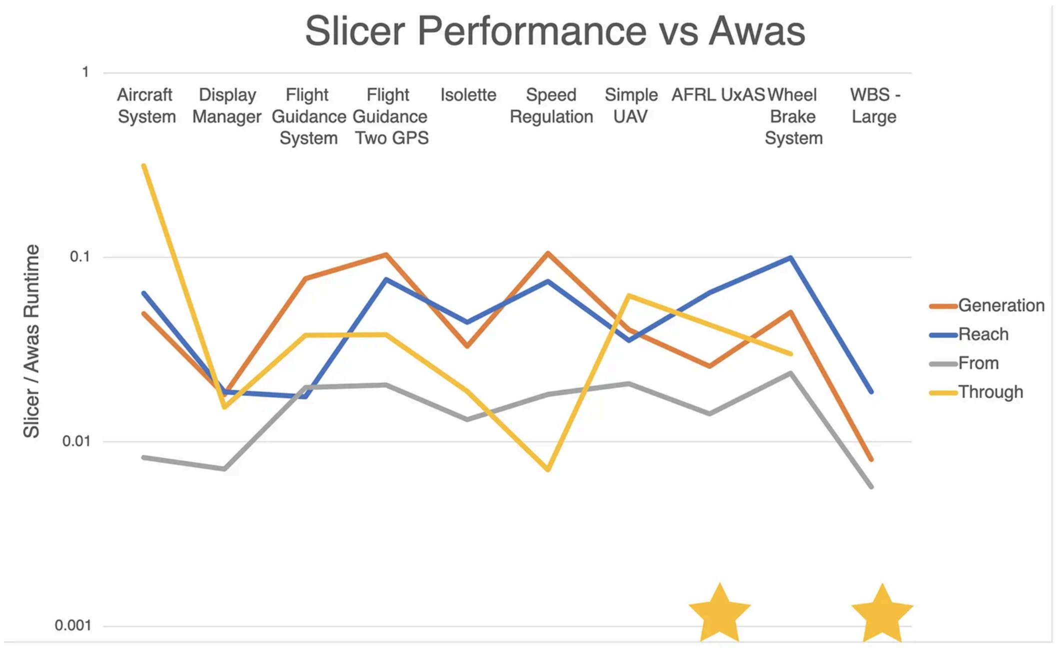


Figure 4: The performance of the OSATE Slicer relative to Awas, not the Y Axis is logarithmic

Overall, we found the performance of the Slicer to be quite satisfactory: we observed a 10-100x speedup over Awas on the generation and querying of nearly all the models in the corpus (see Figure 4). What's more, some reach through queries would not execute under Awas on two of the larger models (denoted with ★ symbols in the figure), but we were able to run them without issue using our tool.

Next Steps: We're Looking for Collaborators!

We're excited about the applications of the OSATE Slicer, both the ones we've identified in this post and those that we haven't even thought of yet. To help us out with those, we're always looking for people to collaborate with—do you have system models that you'd like to analyze more easily or quickly? If so, please reach out. Since their inception, AADL and OSATE have been informed by the needs of DoD and industrial users. The Slicer is no different in this regard, and we welcome user thoughts, feedback, ideas, and collaborations to improve the work.

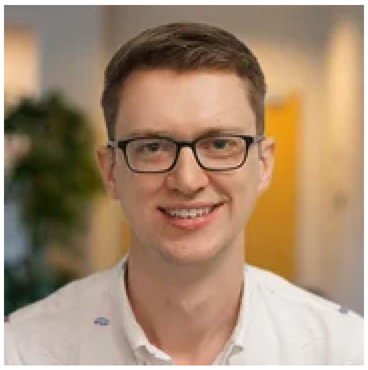
ADDITIONAL RESOURCES

Read "The OSATE Slicer: Graph-Based Reachability for Architectural Models" by Sam Procter.

Learn more about AADL.

Learn more about OSATE.

WRITTEN BY



Sam Procter

DIGITAL LIBRARY PUBLICATIONS

SEND A MESSAGE

MORE BY THE AUTHOR

A Model-Based Tool to Assist in the Design of Safety-Critical Systems

MARCH 7, 2022 • BY SAM PROCTER

Integrating Safety and Security Engineering for Mission-Critical Systems

MAY 10, 2021 • BY SAM PROCTER, SHOLOM G. COHEN

The AADL Error Library: 4 Families of System Errors

MAY 20, 2019 • BY SAM PROCTER

Simultaneous Analysis of Safety and Security of a Critical System

SEPTEMBER 11, 2017 • BY SAM PROCTER

MORE IN SOFTWARE ARCHITECTURE

Building Quality Software: 4 Engineering-Centric Techniques

AUGUST 19, 2024 • BY ALEJANDRO GOMEZ

How to Use Docker and NS-3 to Create Realistic Network Simulations

MARCH 27, 2023 • BY ALEJANDRO GOMEZ

Software Isolation: Why It Matters to Software Evolution and Why Everybody Puts It Off

MARCH 20, 2023 • BY MARIO BENITEZ PRECIADO

Experiences Documenting and Remediating Enterprise Technical Debt

DECEMBER 19, 2022 • BY STEPHANY BELLOMO

What Is Enterprise Technical Debt?

DECEMBER 5, 2022 • BY STEPHANY BELLOMO
