Guided Architecture Trade Space Exploration:
Fusing Model Based Engineering & Design by
Shopping

Sam Procter
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania, United States
sprocter@sei.cmu.edu

Lutz Wrage
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania, United States
lwrage@sei.cmu.edu

Abstract—Advances in model-based system engineering have greatly increased the predictive power of models and the analyses that can be run on them. At the same time, designs have become more modular and component-based. It can be difficult to manually explore all possible system designs due to the sheer number of possible architectures and configurations; design space exploration has arisen as a solution to this challenge.

In this work, we present the Guided Architecture Trade Space Explorer (GATSE), software which connects an existing model based engineering language (AADL) and tool (OSATE) to an existing design space exploration tool (ATSV). GATSE, AADL, and OSATE are all designed to be easily extended by users, which enables relatively straightforward domain-customizations. ATSV, combined with these customizations, lets system designers “shop” for candidate architectures and interactively explore the architectural trade space according to any quantifiable quality attribute or system characteristic. We evaluate GATSE according to an established framework for variable system architectures, and demonstrate its use on an avionics subsystem.

Index Terms—Design Space Exploration, Search-Based System Engineering, Model-Based Engineering, Guided Optimization

I. INTRODUCTION

Construction of large-scale software-based systems is a challenging task, and one that is increasingly expensive. Many modern critical systems, such as aircraft, are compositions of smaller components that are themselves composed of both hardware and software subcomponents. Acceptable behavior of these systems often means meeting strict correctness and timing requirements, which makes them particularly challenging—and thus costly—to build [1]. Additionally, while hardware costs once dominated the development of critical systems, software costs are rising rapidly and are becoming the dominant cost driver [2].

The need to control system development costs has motivated a number of the advancements in related fields. These advancements increase the productivity of designers by, among other things: (a) letting them work at a higher level of abstraction with, e.g., Model-Based Engineering [3]; and (b) finding design improvements semi-automatically, with, e.g., Design Space Exploration [4]. These techniques have led to a number of individual advances but are perhaps most powerful when combined, because they have complementary strengths and offsetting weaknesses.

Model-Based System Engineering Model-based development methods, in which engineers create a model of a system or component and then analyze the model for desired quality attributes, are popular in a range of engineering disciplines. Model based system engineering (MBSE) tools, such as the Systems Modeling Language (SysML, a derivative of UML) [5] and the Architecture Analysis and Design Language (AADL) [3], bring the technique to systems engineering. MBSE has found success by enabling engineers to (a) analyze models for performance in a variety of quality attributes more quickly and cheaply than creating and analyzing full systems, (b) work using graphical tooling that can clearly show various relationships between system components, and (c) test quality attribute performance under potential modifications to rapidly perform “what if?”-style analysis. Though MBSE is useful and has become integral to the development of modern critical systems [3], experience has shown that large models—such as large codebases in programming languages—can become too large for individual developers to easily understand and manipulate. Current tooling offers little help for dealing with the large numbers of component choices, parameter settings, and other design/configuration options present in modern critical systems.

Design Space Exploration Any time a large number of options are available to potentially address some need, search-based techniques are a natural choice. Search-based optimization techniques, Harman et al. note, are not the same as those used to search text; rather they consider problems “in which optimal or near-optimal solutions are sought in a search space of candidate solutions, guided by a fitness function that distinguishes between better and worse solutions” [6]. When applied to design problems, this technique has been termed Design Space Exploration (DSE), and it presents a natural complement to MBSE: not only does the involved fitness function rely on a system model, but the technique can: (a) easily cope with very large numbers of options, (b) be used at all stages of the system development lifecycle [6],
that can be customized, (b) MBSE tooling that can be extended.

In particular, we wanted (a) an MBSE language be equally suitable in the broad range of critical system

In light of the challenges facing systems engineers and the solutions enabled by these two complementary techniques, we created and evaluated the Guided Architecture Trade Space Explorer (GATSE). GATSE connects a DSE tool, Penn State’s ARL Trade Space Space Visualizer (ATSV), and a MBSE tool, the Software Engineering Institute’s Open Source Architecture Tool Environment (OSATE). Specifically, this paper describes the following contributions:

1) The GATSE Software: GATSE enables design space exploration and is embedded in a well-established, industrially used MBSE toolkit. It consists of a configuration language and an extensible plugin to OSATE that enables automated execution and analysis given input from ATSV. Features include:

   a) The ability to develop and integrate user-defined fitness functions (using Java) into OSATE’s “Single Source of Truth” concept of MBSE [9],

   b) Interactive, n-dimensional visualization and guided searching using ATSV [10], and

   c) User-specified constraints on describable system aspects (that are automatically checked for satisfiability).


3) Example: An example demonstrating the use of the tool on a standardized avionics system.

II. MOTIVATION AND VISION

Our goals for this work were to:

1) Synthesize best practices from similar efforts into a modern design-by-shopping tool and supporting domain-specific language.

2) Integrate the tool and language into an existing MBSE toolbench and language.

3) Evaluate our work to determine where there are shortcomings and where we succeeded.

Recognizing that we could not develop a tool that would be equally suitable in the broad range of critical system development efforts, we chose to place a high priority on customizability. In particular, we wanted (a) an MBSE language that can be customized, (b) MBSE tooling that can be extended to support those customizations, and (c) DSE tooling that can explore the trade space of (i.e., graphically display, filter, and adaptively tailor) models built in the language from (a) and the analyses from (b). In order to support the decisions that system architects must make—which grow both in number and complexity as the number of system subcomponents and constraints increases—system design tooling should better organize and clarify tradeoffs. That is, this work does not aim to replace a system architect’s design expertise, but rather to enhance it by providing better, clearer information.

There are a number of tools that support design space exploration for system engineering (see Section VII). A significant opportunity for model and analysis reuse is missed, however, if they require the creation of a) bespoke system models in custom formats, b) similarly customized fitness functions / analyses, or c) worse still, do not support user-specified analyses at all. We decided to extend a well-established MBSE language (AADL, Section III-A) and tool (OSATE, Section III-B) with a configuration sublanguage that could support as many “degrees of freedom” as possible (see Section III-C).

That configuration language is then used to guide system exploration using a well-established DSE tool (ATSV, Section III-D).

The overall vision for this work is shown in Figure 1. The top left code sample, labeled “Custom Properties” is part of the AADL model for a small subcomponent of a larger system. It shows a number of properties which can be analyzed by OSATE, including the domain-specific analysis implemented in the top right code sample. The bottom row of Figure 1 shows different views of an example system’s trade space as plotted by ATSV.

III. BACKGROUND

A. Modeling Language: AADL

The Architecture Analysis and Design Language (AADL) is an internationally standardized architecture description language that was originally released in 2004 [3]. As a language targeted at modeling and analysis of critical systems, its primary constructs are functional and runtime system elements, their interconnections, and properties that attach to both elements and connections. Elements and connections include both hardware and software, e.g., processors, processes, busses, memory, threads, subprograms, etc. [3] AADL specifies a number of standardized property sets, and system designers can also create custom properties from pre-existing or custom property types.

A toy AADL system is shown in Listing 1. It gives a taste for the AADL language, and shows some of the key language elements. The first section, lines 3-8, lists subcomponents of the system, both functional (e.g., process) and runtime (e.g., device) elements made up of hardware (e.g., bus, and memory) and software (e.g., subprogram, not shown in Listing 1). Lines 10-13 show two types of connections: port connections, which are “pathways for . . . directional transfers of [data and events] between components”, and bus access connections, which are
the physical connections between components [3]. The final section, lines 15-17, shows sample properties.

AADL is a declarative language, though most system analyses operate on what is known as the instance model. The process of instantiation converts a declarative AADL model (consisting of element types and implementations, collectively referred to as classifiers) into its instance representation. Instantiating a system will, among other tasks, allocate the system's software elements to the hardware elements they will run on as well as fully resolve all property specifications.

The core language, which describes only the architecture of a system, has been extended by a number of language annexes, including those that enable the modeling of behavioral aspects [13], error propagations and transformations [14], and code generation targeting critical architectures [15].

B. Modeling Toolset: OSATE

There are a number of toolsets, both academic and commercial, that are designed to work with AADL (e.g., AADL Inspector1, CAMET2). The Software Engineering Institute maintains the Open Source Architecture Tool Environment3 (OSATE) which is a customized distribution of the Eclipse IDE. It provides editing support and a range of built-in analyses which rely on the standardized AADL properties. The tool environment is open source and new analyses have been developed by both the OSATE developers and external research groups.

C. Degrees of Freedom in Software Architecture

When creating a design space exploration tool, it can be difficult to determine what should be changeable within a system. While some sources of variability—component choices, variable configuration settings—are obviously necessary, it is not immediately clear what precisely should be changeable and what should be fixed when exploring a system’s design space. We used Koziolek’s list of the 11 Degrees of Freedom (DoF) in models of software architecture [11] as a guide. Though we target different domains (Koziolek focused on a client-server architecture) and use different modeling languages (Koziolek

---

1http://www.ellidiss.fr/public/wiki/wiki/inspector
3http://osate.org
used the Palladio Component Model [16]), we found the theoretical aspects of this work to be very helpful.

Each degree of freedom represents the type of a possible change in a system architecture model: component selection, configuration parameter setting, allocation of software to hardware, etc. Koziolek focused on changes that would affect performance, cost, and reliability, but notes that other quality attributes—such as security—could require other degrees of freedom. We discuss the extent to which we were able to support the DoF identified by Koziolek in Section V-B.

D. Design Space Exploration: ATSV

Penn State developed the ARL Trade Space Visualizer (ATSV) as an “Engineering Decision Making Tool” [17]. It is a graphical tool, designed to be used by system engineers, that helps users visualize the trade space of their systems. It does this by displaying measurable system characteristics in various graphical formats: glyph plots, histograms, parallel coordinates, scatter matrices, etc. [17] As systems can have an arbitrary number of dimensions, only a subset will be viewable in a particular representation: the tool lets the user select which system aspects (i.e., which input and measured values) are displayed.

It can read static data from various file types (e.g., tab or comma-delimited formats) or connect to external models of a system for guided analysis. These models can be in any format or tool environment, as long as the model can be built and analyzed headlessly. Exploration can be guided or focused on arbitrary regions of the design space by specifying preference functions, or “attractor” points, within the design space [10]. Then, as ATSV repeatedly queries the system model, it discerns which inputs affect which outputs using evolutionary algorithms. Other expected functionality, such as determining pareto optimality (i.e., designs where no preferred variable can be improved without worsening another preferred variable) [17] and hiding candidates that are infeasible according to user-specified constraints is present as well [18].

Design By Shopping ATSV was designed to support Balling’s notion of “Designing by Shopping” [7], [12]. Balling argues that optimization techniques which require users to know and be able to quantify all of their preferences before seeing any candidate outputs (what Hwang and Masud term “a priori articulation of preference” [19]) are prohibitively difficult to use. The solution he suggests is to present a number of (ideally pareto-optimal) candidates to users, who can then evaluate the options and explore the tradeoffs between them using their existing engineering expertise. Balling also suggests the need for “interactive shopping tools” that are tailored to the domain [7]. This work, which marries a system modeling tool (OSATE) with software that provides for interactive shopping (ATSV) between pareto-optimal candidates is, we argue, a realization of Balling’s vision in the system design space.

IV. THE GATSE TOOL

The GATSE tooling consists of a collection of modifications to OSATE that (a) support our configuration language, (b) enable headless system instantiation and analysis, and (c) install a small adapter that facilitates communication between OSATE and ATSV. It is used, along with OSATE and ATSV, to interactively explore the design space of a system. We provide an overview of how the tooling is used in the subsection below. We then describe how GATSE is used in two phases: design-time (Section IV-B, where a system’s trade space is specified) and run-time (Section IV-C, where the trade space is explored).

A. Usage

The GATSE tooling is used through the OSATE and ATSV interfaces; there is no standalone GATSE executable. Rather, an open-source\(^4\) plugin is installed into OSATE which contains all the modifications, as well as an installer for the connector and parser for ATSV. A typical use-case is:

1) **User** Begins with:
   a) System model “skeleton” with changeable elements,
   b) One or more (potentially custom) model analyses,
   c) OSATE, ATSV, and the GATSE plugin installed.

2) **User** Specifies (using the configuration language):
   a) What elements are changeable,
   b) The values the changed elements can take,
   c) Configuration constraints (on element selection), and
   d) Output constraints (on model validity).

3) **OSATE** Initializes GATSE tooling:
   a) Verifies element constraints are satisfiable, and
   b) Creates OSATE-ATSV connection artifacts.

4) **User** Triggers design-space exploration.

5) **GATSE** Explores the system’s design space:
   a) ATSV Selects inputs from constrained input space.
   b) OSATE Instantiates model described by skeleton + input values selected by ATSV.
   c) OSATE Runs specified analyses.
   d) **ATSV** Updates:
      i) Adds new values to graphical display, and
      ii) Selects new input values.
   e) **ATSV** Repeats (returns to 5a) until batch size is reached.

B. Design-Time

The use of the GATSE tooling at model-design time mostly revolves around specifying the system model’s configuration: the file that indicates which model elements are changeable, what their possible values are, as well as other data such as the analyses to run, which output variables to measure, and constraints on those outputs. The language that this file is written in is described in Section V.

\(^4\)https://github.com/osate/osate2-gtse
1) Processing the Configuration: Once the prerequisites (i.e., the three parts of Step 0 in Section IV-A) have been met and the configuration file written (Step 1) the user can trigger the GATSE initialization (Step 2). Then, the GATSE plugin for OSATE performs the substeps of Step 3.

First, GATSE must verify that the configuration is not over-constrained, i.e., the user’s configuration constraints do not eliminate all possible system designs. This is done by first translating the constraints into equality logic, and deriving additional constraints from those variables with finite types (i.e., all variables’ types are “baked in” as additional constraints). We note that our inability to check the satisfiability of the portions of a configuration that uses variables with infinite types means that we lose soundness; this is discussed in our plans for future work, see Section VIII-A. We then remove constants from the equality logic using Kroening and Strichman’s algorithm [20] and transform the equality logic to propositional logic using Zantema and Groote’s equality substitution algorithm5 [21]. The propositional logic is then transformed into conjunctive normal form using Tseitin’s transformation [22] and satisfiability is checked using Sat4J [23]. Second, GATSE writes the necessary configuration and auxiliary files to support ATSV-OSATE integration.

2) Integrating new Analyses: In addition to building the system model, designers may want to develop custom, domain-specific analyses. Doing so is straightforward: the GATSE plugin defines an extension point that lets users add new analyses using the standard Eclipse plugin infrastructure. The interface specifies a single required method, which takes as input an instance model of the system and returns as output a key-value store that contains the results of analyzing the model. The code fragment labeled “Custom Analyses” in Figure 1 is the entire implementation of a simple analysis that sums the value of a custom property. However, since analyses have full access to the system model—and are implemented in Java—sophisticated analyses can be implemented as well.

C. Runtime

Once the model has been built and the configuration specification has been processed, the actual design-space exploration can begin. The user initializes this (Step 4 in Section IV-A) and it is here that all elements of GATSE work together.

In addition to the configuration-processing functionality described in Section IV-B1, the GATSE plugin for OSATE also installs hooks into OSA TE’s instantiation logic. Recall from Section III-A that one purpose of instantiating AADL is to fully allocate software elements (e.g., processes, port connections) to hardware elements (respectively: processors, buses). The modified version of the instantiator replaces the model’s original elements as they are encountered with the versions selected by ATSV, fixing any connections that relied on the original elements. Once the model has been built, when property values would normally be determined by a

2To our knowledge, ours is the first open-source implementation of their algorithm.

Listing 2: A toy configuration for the system from Figure 1

```
1 Root Complete
2 configuration Complete {
  3 spd_snsr_opts: device Sample::snsr from
    4 (Sample::snsr.spd_cheap, Sample::snsr.spd_mid, Sample::snsr.spd_quality),
  5 data_rate_opts: Communication_Properties::Data_Rate from
    6 (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024)
  7 extends Sample::Complete.PBA_spd_ctrl {
    8 spd_snsr => spd_snsr_opts,
    9 DC1#Communication_Properties::Data_Rate => data_rate_opts
  10 constraints {
    11 spd_snsr == !Sample::snsr.spd_cheap requires
      12 DC1#Communication_Properties::Data_Rate in {1, 2, 4, 8},
    13 spd_snsr == !Sample::snsr.spd_mid forbids
      14 DC1#Communication_Properties::Data_Rate in {256, 512, 1024}
  15 analyses {
    16 org.osate.atsv.integration.property-totals
  17 outputs {
    18 ValModel : float,
    19 InvalidReason : string,
    20 Price : float < 500.0,
    21 Weight : float < 3000.0
  22}
```

V. THE CHOICEPOINT LANGUAGE

In this section we describe the configuration language we created to describe a system design’s trade space. We first provide a walkthrough of the language, and then an evaluation against the set of capabilities identified by Koziolek [11].

A. Walkthrough

An example configuration file, corresponding to the system from Listing 1 and containing only one configuration specification, is shown in Listing 2. Lines 3 and 4 are the specification’s parameters (see Section V-A1), lines 5-7 are the extends clause (see Section V-A2), and lines 8-10 are constraints (see Section V-A3). Note that a full configuration file would likely contain many individual specifications (i.e., groups of parameter, extends, and constraints sections) as well as multiple analyses and more outputs.

1) Parameters: A configuration’s parameter specification lists both what is changeable in an element and what the options for those changes are. It consists of a name for the set of changes, the type of the changes (an AADL classifier or property), and then the set of allowed values. For example, line 3 of Listing 2 specifies that there are three options for the speed sensor in the hypothetical system from Listing 1.
Line 4 shows the second parameter for the configuration, in this case the data rate that the sensor can be configured to transmit at. Note that while we explicitly enumerate the options in this example, if the range of allowable data rates was contiguous, we could have simply specified the maximum and minimum values (i.e., \((1 .. 1024)\)) to allow selection of any in-range number.

2) Extends: A configuration’s \texttt{extends} clause specifies which elements of the skeleton model the configuration applies to (line 5 in Listing 2) and then maps the parameters specified previously to the subcomponents and properties in the element itself (lines 6-7). Elements are referenced using their qualified\(^6\) path through the instance model, and properties are referenced using the \# character. In some cases, it may be necessary to vary both the type and implementation of a component, and this can be done using a \texttt{with} statement (not shown in Listing 2), \texttt{with} statements are a more general purpose construct, though, that lets designers combine multiple configuration specifications for the same element.

3) Constraints: The third section of a configuration specification specifies any constraints on the selection of values. These may be necessary if certain components cannot function together (due to, e.g., software incompatibilities, physical requirements, etc.) or because some options cannot support a subset of configuration values. This is the case in Listing 2: the lower quality and less expensive sensors cannot support higher data rates (lines 9-10). Six types of constraints are supported; Table I gives their syntax and informal semantics.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A == B)</td>
<td>The value of element (A) must be equal to the value of element (B)</td>
</tr>
<tr>
<td>(A != B)</td>
<td>The value of element (A) must not be equal to the value of element (B)</td>
</tr>
<tr>
<td>(A == X) requires (B == Y)</td>
<td>If the value of element (A) is (X) then the value of element (B) must be (Y)</td>
</tr>
<tr>
<td>(A == X) forbids (B == Y)</td>
<td>If the value of element (A) is (X) then the value of element (B) must not be (Y)</td>
</tr>
<tr>
<td>(A == X) requires (B) in ([Y, Z])</td>
<td>If the value of element (A) is (X) then the value of element (B) must be (Y) or (Z)</td>
</tr>
<tr>
<td>(A == X) forbids (B) in ([Y, Z])</td>
<td>If the value of element (A) is (X) then the value of element (B) must not be (Y) or (Z)</td>
</tr>
</tbody>
</table>

TABLE I

CONSTRAINT SYNTAX AND SEMANTICS.

Collectively, these six pieces of information describe the Degrees of Freedom, such as “Selection of components,” (see example on line 3 of Listing 2), and “Non-functional Component Configuration Param,” (see line 4 of Listing 2) [11]. Note that we have not established a formal mapping between Koziolek’s formalizations and our own work as hers were based on the Palladio Component Model (PCM) [16]; a mapping between AADL and PCM is beyond the scope of this work. We are satisfied, though, with the capabilities of our language insofar as (a) it enables the use case described in the following section, (b) it supports most of the information required by Koziolek’s Degrees of Freedom, and (c) it compares favorably with existing work (see Section VII).

VI. USE CASE: A WHEEL BRAKE SYSTEM

We used a model of a fictional aircraft wheel brake system (WBS) as a case study. This system is fairly well-studied in the critical-system and model-based engineering literature; it was originally created as part of the ARP4761 and ARP4754 standards, and has been described and (re)modeled as part of a number of efforts since then [24]–[27].

We selected the WBS primarily because it is relatively straightforward but still demonstrates many of the complexities
in modern system design. A number of these complexities are particularly relevant for our work on this effort, including: (a) multiple candidate architectures, (b) redundant components, and (c) shared interconnections relied on by heterogeneous components. We also selected the WBS because of a more general trend in avionics systems towards modularity and a component basis. The introduction of component-based architectures into aircraft has led to important benefits, and has expanded into software development with technologies like the Integrated Modular Avionics architecture [28]. As more hardware components and software modules become available, it becomes increasingly challenging to understand which combination of them is best given a large set of desired—and to some extent competing—quality attributes such as cost, power consumption, latency / performance, weight, and efficacy.

The full system model, GATSE and ATSV configuration files, and generated results are open-source and publicly available.

A. System Description

A simplified view of the WBS architecture is shown using AADL’s graphical notation in Figure 2; note that some of the hardware elements relied upon by the system are not shown in this view. The elements that make up this portion of the architecture, reading roughly left-to-right, are [27], [30]:

1) Pedals: The pilot’s brake pedals, which indicate the desired amount of braking power.
2) Power: A power supply with redundant connections to the BSCU.
3) Brake System Control Unit (BSCU): A collection of software that controls the braking of the aircraft. It is responsible for controlling the anti-skid, selector, and shutoff valves.
4) Pumps: Hydraulic pumps which provide the pressure necessary for braking. In normal operation, the green pump is used; the blue pump is an alternate.
5) Accumulator: A gas-powered emergency source of hydraulic pressure. Used when both the green and blue pumps have failed.
6) Shutoff Valve: A valve to disable the green pump if the BSCU determines the system should stop using it (due to, e.g., insufficient pressure).
7) Selector Valve: A valve that selects a source of pressure (based on input from the BSCU) and applies it to the skid valves.
8) Anti-Skid Valves: Valves that control hydraulic pressure to the brakes, and limit it so the wheels do not lock.
9) Wheel: A wheel and brake assembly.

The full model includes a number of simplified subsystems (hydraulic, electrical, alert, etc.) that are not shown in Figure 2.

### Table II

<table>
<thead>
<tr>
<th>Component Interconnections</th>
<th>Options</th>
<th>Red?</th>
<th>Const?</th>
<th>Type?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power bus</td>
<td>6</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Property Specifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Power</td>
<td>∞</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

B. Trade Space Exploration

As described in Sections IV-B and IV-C, use of GATSE consists of two phases: design-time activities which specify the trade space, and run-time activities which explore it.

Listing 3: A snippet of the full WBS configuration, specifying choices and constraints for the CPUs used in the federated architecture.

```plaintext
configuration platform_federated_conf {
2 power_budget : SEI::PowerBudget from (0.1W .. 300W),
3 cpu_arch : processor impl::platform::cpu from
4 cpu_arch : processor impl::platform::cpu.arm
5 extends impl::platform::platform.federated |
6 cpu2.power#SEI::PowerBudget => power_budget,
7 cpu1 => cpu_arch with cpu_base_conf,
8 cpu2 => cpu_arch with cpu_base_conf
9) constraints {10 cpu1 == cpu2
11}
```

1) Design-Time: Trade Space Specification: For the purposes of testing, we created two or three options for several WBS classifiers (e.g., component types, implementations, hardware interconnects), and affixed new properties. Examples of these are given in Table II, and a small section of the full configuration file is shown in Listing 3. Even in our relatively small example system, a brute force enumeration of the possible candidate architectures quickly became infeasible: the trade space of the WBS system includes hundreds of millions of different configurations.

The number of feasible choices, while still quite large, was restricted significantly by constraints we created. We attempted to create a number of interesting and realistic constraints, including:

1) Hardware Restrictions on Software: We require that certain deceleration hardware assemblies require the use of certain BSCU command software.
2) Power Sources Restricting Wire Gauge: Some power sources were modeled to be more powerful than others; we disallowed connections to those larger sources using thinner wiring.

https://github.com/osate/osate2-gtse/tree/master/GTSE-Examples
3) **Identical CPU Architectures within Assemblies:** In a federated architecture, we required that the CPUs used the same architecture.

We note that it is possible to accidentally overconstrain the model so that no valid architectures could be generated. The satisfiability checking described in Section IV-B1 successfully detected and warned us of these errors.

We enabled a relatively small number of system analyses for our initial searches. Our expectation is that the number and power of selected analyses will increase as a system’s design trade space shrinks: early on, it is more important to be able to rapidly enumerate multiple candidate architectures and evaluate them relatively quickly. That is, the fitness function used should initially be relatively inexpensive; later on more expensive calculations can be used for finer grained analyses. Specifically, we checked each candidate architecture’s: (a) weight, (b) price, (c) power consumption, (d) port consistency (to verify that candidate architectures did not have mismatched connection types), and (e) “braking power.” This final analysis type is not a true analysis, but rather was created to demonstrate the ease with which domain-specific analyses can be created and used with GATSE. In our case, it was a simple summation of property values that had been added to components, but as analyses are implemented using Java, there is considerable flexibility for more sophisticated techniques. The top-right portion of Figure 1 shows the complete implementation of the braking power analysis.

2) **Run-Time: Trade Space Exploration:** Recall from Section II that our goal is to make clear to the system architect the correlations and tradeoffs between a system’s various quality attributes. The design-by-shopping paradigm suggests that this knowledge be used—ideally in an interactive, graphical tool—to refine a system’s design parameters and ultimately select a candidate system architecture [7].

The three boxes in the lower half of Figure 1 are screenshots of ATSV after having explored the WBS trade space in various ways. Of particular interest are the following activities:

1) **Viewing:** ATSV and GATSE lets designers view the system design trade space in a number of formats, e.g., the scatter plot in the lower left of Figure 1 as well as parallel coordinate and histogram plots (not shown). These views are highly customizable; any input or output value can be used for the plots’ axes (e.g., for the scatter plot: X, Y, point color, and point size). Note that output values can be both quantitative measures (e.g., various quality attributes), as well as measures of validity (e.g., feasibility of construction).

2) **Filtering:** While GATSE supports flagging generated architectures as invalid if measured outputs fail to meet some standard (e.g., if the price or weight exceed given thresholds, see lines 18-19 of Listing 2), it is also possible to filter the views directly in ATSV, as in the lower middle portion of Figure 1.

3) **Tailoring:** Though ATSV offers a number of mechanisms for specifying preferences, we found the most success using the Multi-Objective Evolutionary Algorithm (MOEA) Pareto Sampler. It can be configured, using the interface shown in the lower right of Figure 1, to prefer low or high values for either input or measured variables. Pareto optimal designs can also be highlighted. Figure 3 shows the WBS trade space with pareto-optimal architectures marked and a curve fit to those points.

4) **Investigating Specific Candidates:** When selected, the
points in the scatter plot (and lines in the parallel coordinate plot) display the complete set of the input and output values for their associated candidate architecture. This lets designers see the exact configuration of a candidate architecture, as well as exact values of the results of the system analyses and other outputs. Figure 4 shows an example of one candidate architecture for the WBS.

Though performance was not considered a high priority in the development of GATSE’s initial prototype, we note that usability would be improved by decreasing the time it takes to identify and analyze a given number of candidate architectures. Identifying, instantiating, and analyzing a single system architecture takes roughly 3.2 seconds; generating the thousand architectures used for Figure 1 took roughly 53 minutes. We explored the tradespace of the WBS on a machine with an Intel i7 CPU running at 3.4GHz and 32 gigabytes of memory. We used OSATE 2.3.4.vfinal, GATSE 1.0.0.201808031833, and ATSV 10.0.8.32bit.

VII. RELATED WORK

The idea of using search tools to explore the design space of system architectures has been considered before, and previous work in this area has made a number of important contributions. We survey some of the most relevant work in this section, but note that the works cited in this section, in particular Ross et al.’s [35], contain useful lists of related work. An overview of the feature sets of the related work in this section is given in Table III. The programs varied considerably in both goals and implementation strategies, though, so the values in Table III are necessarily somewhat subjective.

The most immediately relevant prior work is ArcheOpterix, a tool described by Aleti et al. [31]. It was an OSATE plugin that used an evolutionary algorithm to guide system architects with deployment decisions, i.e., which task should be allocated to which processor. While similar to GATSE in many ways, it did not feature the robust visualization or evolutionary algorithm options of ATSV, supported a smaller set of choicepoint types in AADL, and the flexibility of analysis specification is unclear. Additionally, the tool’s website is no longer available, and it was developed for a previous version of AADL and OSATE.

Kerzhner developed a language for representing “Architecture Exploration Problems” that is based on SysML [32]. Particularly valuable is the discussion of the search process; it identifies many of the core tradeoffs of the domain. These include the need for visualization and the tradeoff between performance, analysis accuracy, and trade space size. This work uses a collection of mathematical statements for system performance (rather than our more programmatic analyses) relies on custom extensions to SysML that do not describe software or controllers, and is not fully interactive (and thus has less support for design by shopping).

Iacobucci developed the “Rapid Architecture Alternative Modeling” (RAAM) methodology for exploring a system’s trade space during early concept design [33]. It places a large emphasis on performance (by, e.g., implementing parallelization and reducing model complexity) and scalability. Architectures are described using the US Department of Defense’s Architecture Framework (DoDAF) [37]. A domain specific language was used to describe system capabilities, and then during a generation step, the RAAM tooling enumerates possible system architectures. DoDAF is much higher level than AADL, though, and so is not sufficient for our goals of low-level actual system architectures. As with Kerzhner’s work, it is not clear that RAAM is interactive enough to allow users to iterate and refine their searches easily.

Eder and Voss developed AutoFOCUS3 and use it for exploring a system’s architectural design space [4]. This work assumes an analyzable model (e.g., SysML [5]) and then requires the user to specify constraints and objectives using their tool’s interface. The constraints are then discharged to a SMT solver (the authors use Microsoft’s Z3 [38]), and if satisfiable, the various architectures can be compared graphically. Compared to GATSE, AutoFOCUS3 does not generate candidate architectures (though it generates deployment plans and task schedules) and its visualization techniques are targeted at relatively small numbers of systems.

Adventium Labs has developed a Design / Trade Space Explorer that also uses AADL, OSATE, and ATSV [34]. It
takes as input a spreadsheet listing component options and property values, which correspond to component specifications in an AADL file, rather than using a domain specific language describing system design options as in GATSE. Component implementations are selected manually (properties can be randomly selected from within specified ranges) and once all component types have an implementation selected, the system can be instantiated. Alternatively, instances can be automatically enumerated based on component options, though the authors note that this can be very expensive for examples that are not small. Analyses are then run manually (individually or in batches) on the generated instance, after which results get collected into a file. The process is then repeated, and the results file can be loaded into ATSV and explored visually. Compared to our approach, Adventium’s supports more features of AADL, notably annexes. It does not support constraints between components, though, and crucially system creation and analysis are done manually after which ATSV is run on static data—rather than the dynamic, interactive approach enabled by GATSE.

Ross et al. present a language and tooling for exploring automotive architectures hierarchically and from a number of perspectives [35]. This work uses the modeling language Clafer [39], and contains many of the same analyses that are used in OSATE and GATSE. Their work does include a comparison to OSATE, which correctly notes the lack of architecture variability and support for optimization / constraints, both of which have been added by GATSE and ATSV. Additionally, it is unclear how easy it would be for a domain specialist to add new analyses, or extend existing ones.

Esfahani et al. present GuideArch [36], which uses fuzzy math [40] to enable a representation of the ambiguity endemic to early system designs. They present a formalization that enables the comparison of architectures with various ambiguous values under various weightings and constraints. They implemented their formalization in a web-based tool, and discussed positive and negative aspects discovered during an evaluation. There is no integration with other system modeling languages or programmatic analyses, and designers need to know the relative importance of system features a priori. Additionally, the interaction between constraints and fuzzy numbers may over-restrict system designs: a constraint is considered violated if it is possible its value exceeds the constraint’s limits.

VIII. DISCUSSION AND FUTURE WORK

In Section II, we laid out three goals for this effort—to synthesize best practices, integrate our new tooling and language into existing work, and evaluate the results. Overall, the effort was a qualified (see Section VIII-A) success. It was a success in that, we argue, GATSE represents an advancement over the state-of-practice as it includes many of the features found in similar projects (see Section VII), and also enables design-by-shopping [7] in a standardized yet highly-customizable system design language and tool. By writing the configuration language to work with AADL (which is widely used in critical system development) and our tools with OSATE (the standard workbench used to develop AADL specifications) there is significant opportunity for model reuse beyond GATSE. What’s more, as AADL and its analyses are largely compositional, wholesale adoption of GATSE is not required for benefits to begin to accrue: defining only a small number of choicepoints and exploring potential architectures visually could still be advantageous given a large design space. Finally, the evaluation of the configuration language in Section V-B both demonstrates the positive aspects of the language and highlights aspects that require further work.

A. Future Work

Some aspects of this work will likely need to be improved before it is more widely used, in particular the complexity and performance aspects.

1) Change Specification Complexity: In modeling even toy systems, and especially when building the WBS example, it became clear that specifying changes in a hierarchical architecture can be quite challenging. That is, one set of changes may depend on a second set, which may depend on a third set, etc. Our language could be improved to help manage this complexity. Our constraints are fairly powerful, but can be difficult enough to use that syntax improvements, more advanced features like aspect orientation [41], or even generation from a higher-level language would be worth investigating. There are also research efforts attempting to create a metamodel for design space exploration, we are interested in seeing the degree to which we could use their concepts to improve our language [42].

2) Performance: OSATE is designed for a single user following a traditional design methodology, rather than being given automatically-generated specifications and continually re-instantiating and re-analyzing them. We believe that design space exploration becomes more useful the more quickly architecture candidates can be generated and analyzed—an argument made by a number of others, including Iacobucci [33]. We would like to see the extent to which OSATE can be optimized so that it can be more time efficient.

3) Adding / Removing System Components: Koziolek’s degrees of freedom [11] include one—adding and removing components from a system—that GATSE cannot support. We are looking into what would be required to support this functionality, but it is far more complex than the other degrees of freedom, and adds considerably to the change specification complexity discussed previously.

4) Soundness Checking for Constraints: As discussed in Section IV-B1, our current implementation of satisfiability checking for constraints can only operate on a subset of the property types used in AADL. We would like to improve GATSE to remove this restriction. If we were to switch from our current SAT solver (Sat4J [23]) to a SMT solver such as Z3 [38], we expect to be able to operate on a much larger subset of AADL’s property types.

IX. CONCLUSION

In this work we described GATSE: a new extension to the OSATE toolset that, by adapting it to work with ATSV, enables
interactive exploration of a system’s architectural design space. System designs can be analyzed and constrained using a number of different quality attribute measures, and new, domain-specific analyses can be created in a straightforward manner as well. We evaluated our work against an established theoretical framework from Koziolke [11] and demonstrated it on a case study from the avionics domain.

ACKNOWLEDGEMENTS

The authors wish to thank Julien Delange, Min Young Nam, and Peter Feiler for the original concept and their feedback. We also gratefully acknowledge the assistance of Gary Stump and Penn State University’s Applied Research Laboratory for their help and the modifications made to ATSV as a result of this effort.

© 2019 IEEE and Carnegie Mellon University. All rights reserved. This is the pre-publication version of a paper expected to be published in 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS). 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. DM18-1401.

REFERENCES


