Noname manuscript No.
(will be inserted by the editor)

Guided Architecture Trade Space Exploration

Fusing Model Based Engineering & Design by Shopping

Sam Procter · Lutz Wrage

the date of receipt and acceptance should be inserted later

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; trade space exploration has arisen as a solution to this challenge.

In this work, we present a new software tool: the *Guided Architecture Trade Space Explorer* (GATSE), which connects an existing model-based engineering language (AADL) and modeling environment (OSATE) to an existing trade 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.

Keywords Design Space Exploration · Search-Based System Engineering · Model-Based Engineering · Guided Optimization · Architecture Analysis and Design Language (AADL) · Open Source AADL Tool Environment (OSATE) · ARL Trade Space Visualizer (ATSV)

This version of the article has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: `http://dx.doi.org/10.1007/s10270-021-00889-8`

Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States E-mail: sprocter@sei.cmu.edu, lwrage@sei.cmu.edu

1 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 [12]. Additionally, while hardware costs once dominated the development of critical systems, software costs are rising rapidly and are becoming the dominant cost driver [11].

The need to control system development costs has motivated a number of advancements in related fields. These advancements increase the productivity of designers by, among other things: letting them work at a higher level of abstraction with, e.g., *Model-Based Engineering* [21]; and finding feasible design candidates and improvements semi-automatically, with, e.g., *Design Space Exploration* (also known as *Trade* Space Exploration) [17]. These techniques 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 Lan-*

guage (SysML, a derivative of UML) [24] and the *Architecture Analysis and Design Language* (AADL) [21], 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 [21], experience has shown that large models—much like 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.

When engineers using MBSE approaches speak of evaluating design alternatives, they are typically referring to a manual process that involves creating a model, analyzing various characteristics about it, and trying to learn the costs and benefits of the tradeoffs that can be made. This process does not scale as the number of component and configuration options increases since the number of combinations rises too rapidly to be analyzed manually. A key insight motivating this and related work is that that manual process mimics automated techniques known as design space exploration.

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" [25]. 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:

- easily cope with very large numbers of options,
- be used at all stages of the system development lifecycle [25],
- be tuned to sample broadly from the global search space or narrow portions using increasingly sophisticated sampling algorithms, and
- be further improved by user-interaction (as opposed to completely automatic optimization). [5]

Without user interaction, search becomes the well-studied problem of optimization; the "best" system, according to a fitness function, can often be determined with relatively little engineering effort. But knowing, a priori, the correct relative weights for various quality attributes is prohibitively difficult [5]. Additionally, many DSE tools—even those that are designed specifically for system architecture evaluation—use purpose-built, purely mathematical fitness functions (e.g., [43]) that are not automatically derivable from system models. These tools are very specialized, which may impact adoption: what would be more usable would be a DSE tool that can leverage modeling languages, tooling, and automated system analyses that are already in wide-scale, industrial use.

In light of the challenges faced by systems engineers and the solutions enabled by these two 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 [22],
   (b) Interactive, n-dimensional visualization and guided searching using ATSV [48], and
   (c) User-specified constraints on certain system aspects that are automatically checked for satisfiability.
2. **The Configuration Language:** A language for describing system and software choicepoints and constraints mapped to an existing DSE-capability framework [31].
3. **Example:** An example demonstrating the use of the tool on a standardized [45, 46] avionics system. The example is supported by two new system analyses that illustrate the phases of design space exploration in GATSE.

This paper is an extended version of a conference publication that was presented at MODELS 2019 [36]. It differs from the conference paper by including:

- A description of how GATSE can be used for multi-phase design space exploration including a concrete

example of, and conceptual elaboration on, quantifying system safety / security via customized analyses;
- Significantly more detailed process description, use case, and evaluation sections;
- A clarified motivation, which is derived from an expanded and more detailed related work section, against which the evaluation is now performed; and
- A significant number of new figures and clarifications throughout.

The remainder of this work is organized as follows. We review related work in Section 3, and use that to inform the motivation for this work in Section 2. We discuss relevant background material in Section 4, the GATSE tooling itself in Section 5, and the configuration language in Section 6. We provide a use case and discussion in Section 7, and evaluate our work in Section 8. We then provide a roadmap for future work in Section 9 and conclude in Section 10.

2 Motivation

We were motivated to see how state-of-the-art work in DSE could be exploited within the critical system space. 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 recognized the value of prior works that had: (a) an (ideally standardized / widely used) MBSE language that can be easily customized, (b) MBSE tooling that can be extended to support those customizations in its analyses of system models, 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) using the analyses from (b).

2.1 Six Objectives

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—our overall aim was to produce a tool-supported, model-based DSE process which achieved six high-level objectives. These objectives were identified from a review of relevant literature; various previous approaches in this domain showed the value of these criteria.

1. **Expressive Configuration Language:** The language used to specify system design choices should be powerful enough to describe the wide variety of feasible system changes, instead of being restricted to, e.g., deployment strategies or performance optimizations. Koziolek's work [31] identifies what should be expressible, and how it should be expressed. We aim to support as many of the parameters necessary to describe what she terms a system's *Degrees of Freedom* as possible.
2. **Easily-Used Configuration Language:** The configuration language must also be relatively easy to use by system designers.
3. **Tool Support for Modeling:** Any process we propose should have tool support for developing the system models that will be analyzed.
4. **Re-usable Modeling Language:** Ideally, models built for design-space exploration could be re-used in other MBSE efforts, rather than being created for the DSE effort and then discarded.
5. **Support for Extensible Analyses:** Due to the variety of domain-specific requirements, new types of analyses should be easy to develop by system modelers. Unfortunately, these analyses *must* be quantitative, as there is no way to automatically compare arbitrary qualitative data.
6. **Support for Design-by-Shopping:** At a minimum, our approach should be interactive and iterative, and ideally support a graphical, design-by-shopping paradigm of system design. This objective requires a high level of performance, since long delays will decrease the level of interactivity.

2.2 Two Challenges

In addition to the six criteria mentioned above, there are two challenges that all approaches in this domain must reckon with. These are not solvable problems, but rather impediments that any automated tradespace exploration tool / process must address. We aimed to use GATSE to explore approaches to meeting these challenges, but as they are higher-level, evaluation against them is necessarily more subjective. While these challenges impact our criteria, particularly criteria 5 and 6, addressing the challenges is necessary but not entirely sufficient for meeting the specified criteria.

Quantification A potential weakness of most DSE techniques is that they can only examine tradeoffs between system characteristics that can be quantified. Quantifying traditionally qualitative, yet extremely important, characteristics such as a system's safety or security thus gains increased importance. The wisdom of pursuing quantitative approaches has been criticized, see, e.g., Owens and Leveson on Safety [34] or Verendel on Security [52]. We do not aim to defend or advocate particular approaches in this work; we merely note

the necessity of using quantitative data in DSE more broadly and our approach specifically.

Performance Clarifying and organizing system design tradeoffs is challenging because it requires satisfying two goals: the number of options presented to the analyst should be large, and the information about those options should be rich. Unless time or computational resources are effectively unlimited, however, these goals conflict: decreasing the time spent analyzing each system design candidate increases the number of options presented but decreases the depth of information available to the analyst. Similarly, running more analyses (or slower, more sophisticated analyses) on system models will reduce the number of options presented to system architects.

3 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 [38], contain useful lists of related work. An overview of the feature sets of the related work in this section is given in Table 1. The programs varied considerably in both goals and implementation strategies, though, so the values in Table 1 are necessarily somewhat subjective.

3.1 Tools using Standardized Modeling Notations

Many approaches relied on standardized modeling notations as a basis for the system models. While this imposes additional constraints compared to a more abstract approach, it firmly grounds the techniques in the domain targeted by the modeling language and enables easier model re-use.

3.1.1 AADL – Aleti et al.: ArcheOpterix

The most immediately relevant prior work is ArcheOpterix, a tool described by Aleti et al.[3]. It is 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. The tooling and modeling language used give it a broad applicability to a range of systems, and—by reusing a common model-based engineering language—did not require the creation of custom models. While its specification format is powerful, it focuses primarily on the allocation of software elements to hardware resources rather than the more broad set of possible system design choices.

3.1.2 AADL – Adventium Labs: AFFMAD

Adventium Labs has developed the Architecture Framework for Fault Management Assessment and Design (AFFMAD), a Design Space Explorer that also uses AADL, OSATE, and ATSV [2]. It takes as input a spreadsheet listing component options and property values, which correspond to component specifications in an AADL file. 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. Note that since these analyses are OSATE plugins, they are user-extensible, which makes the tool usable for a range of settings, including those with domain-specific requirements. After executing the tool, the process can then be repeated to collect more results, after which the results file can be loaded into ATSV and explored visually. The resulting process is very manual: i.e., it lacks any feedback from ATSV, which is used exclusively for visualization; its evolutionary algorithms are not used to guide the exploration of the design space.

3.1.3 SysML – Kerzhner's Architecture Exploration Language

Kerzhner developed a language for representing "Architecture Exploration Problems" that is based on SysML [29]. 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 to describe system performance (instead of, e.g., a more general analysis framework), relies on custom extensions to SysML that do not describe software or controllers, and is not fully interactive. Kerzhner discusses the need for flexibility in the formulation of system specifications (see Section 1.4.4 of [29]), and while his custom language is expressive, it does not map cleanly to a system's architecture.

	ArcheOpterix [3]	AFFMAD [2]	Kerzhner [29]	AutoFOCUS3 [17]	RAAM [28]	Ross et al.[38]	SQuAT-Vis [23]	GuideArch [19]	Hegedüs et al.[26]	Abdeen et al.[1]
1. Configuration Language Expressiveness	◐	◐	●	●	○	●	◐	●	●	●
2. Ease of Specifying Configuration	○	◐	○	●	●	●	◐	○	◐	◐
3. Tool Support for Modeling	●	●	◐	◐	◐	●	●	●	◐	◐
4. Standard Modeling Language	●	●	◐	◐	●	●	●	○	○	○
5. Extensible Analyses	◐	●	○	○	○	◐	●	◐	◐	◐
6. Interaction / Support for Design-by-Shopping	○	◐	◐	●	◐	●	●	○	○	○

Table 1 Overview of related work. ● signifies full, ◐ partial, and ○ poor or no support for a given feature.

3.1.4 SysML – Eder and Voss: AutoFOCUS3

Eder and Voss developed AutoFOCUS3 and use it for exploring a system's architectural design space [17]. This work assumes an analyzable model (e.g., SysML [24]) and then requires the user to specify constraints and objectives using their tool's straightforward graphical interface rather than a custom language. Because of the higher level of abstraction, the range of possible constraints and objectives is quite broad. The constraints are then discharged to a SMT solver (the authors use Microsoft's Z3 [33]), and if satisfiable, the various architectures can be compared graphically. The visual nature of the comparison makes the tool easier to use than those that are strictly text-based, however it is targeted at relatively small numbers of candidate architectures and may be harder to use with larger trade spaces.

3.1.5 DoDAF – Iacobucci: RAAM

Iacobucci developed the "Rapid Architecture Alternative Modeling" (RAAM) methodology for exploring a system's trade space during early concept design [28]. Iacobucci explains the need for, and places a large emphasis on, performance and scalability by, e.g., implementing parallelization and reducing model complexity. Architectures are described using the US Department of Defense's Architecture Framework (DoDAF) [16], a choice that (compared to the use of a custom architecture description language) made his methodology applicable to a wider range of systems. A custom language was used to describe system capabilities, however, then during a generation step the RAAM tooling enumerates possible system architectures.

3.1.6 Clafer – Ross et al.

Ross et al. present a language and tooling for exploring automotive architectures hierarchically and from a number of perspectives [38]. By using the modeling language Clafer [4], they avoid the pitfall of requiring creation of custom system models. Additionally, their tool contains many of the same analyses that are used in OSATE and this work. 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 this work). It is unclear how easy it would be for a domain specialist to add new analyses, or extend existing ones.

3.1.7 PCM – Frank and van Hoorn: SQuAT-Vis

Frank and van Hoorn describe SQuAT-Vis [23], a visualizer primarily designed for SQuAT [37], which is a software architecture optimizer primarily designed to work with architectures described using the Palladio Component Model (PCM) [6]. The choice to build on existing work for optimization (which builds on an extant modeling language) serves the effort well: as there is already high-quality tool support and analyses for PCM, the authors were able to spend their effort focusing on the user experience. They note that existing approaches to architecture optimization provide limited visualizations, and comment specifically that ATSV requires "a basic understanding of visualization... and [is] unable to visualize software architectures"—we seek to address the latter limitation, to some extent, in this work. Their tool's capabilities include the selection of candidate architectures, stopping criteria, and implementations, as well as a strong focus on visualizations. SQuAT relies on software agents (known as *dBots*) that aim to optimize specific utility functions, which leads to a negotiation-based approach to optimizing system designs. PCM is primarily targeted at performance assessment, though examples of tactics seeking to improve modifiability are also discussed [37].

3.2 Tools without Standardized Modeling Notations

These tools do not use a standardized modeling notation, and as a result have more abstract, and potentially widely-applicable, approaches.

3.2.1 Esfahani et al.: GuideArch

Esfahani et al. present GuideArch [19], which uses fuzzy math [55] to enable a representation of the ambiguity endemic to early system designs. They present a formalization that enables the comparison of architectures with ambiguous values (e.g., the authors mention that a particular feature is expected to use 10μJ of battery, but may use as little as 8μJ or as much as 14μJ) under various weightings and constraints. They implemented their formalization in a web-based tool, and discussed positive and negative aspects discovered during an evaluation. Designers need to know the relative importance of system features a priori, however, which may be challenging. 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.

3.2.2 Hegedüs et al.

Hegedüs et al. describe a very flexible approach to design space exploration that treats the problem as a one akin to exploring a graph of model transformations, where an initial candidate repeatedly has small modifications applied to it [26]. Their work describes the problem space in somewhat similar terms to ours: both grapple with exploration of design spaces that are too large to explore completely. Their technique involves applying classic model-checking-style optimization approaches including pruning of unpromising branches, vectors containing steps needed to transform the initial candidate into the final one, and the encoding of hints to guide exploration. The configuration can be somewhat complex, though the authors describe a software tool they have created to aid system designers in specifying their inputs using a flexible query-evaluation framework.

3.2.3 Abdeen et al.

Abdeen et al. describe a system that is in many ways similar to that of Hegedüs et al., with the exception that exploration is guided by the NSGA-II evolutionary algorithm [13]. Like Hegedüs et al., their implementation uses a generic modeling framework for system models and a generic query framework for evaluating the fitness of those models. This approach is quite flexible making it more broadly applicable but potentially adding a learning curve to use by non-experts. Interestingly, their problem statement differs from most in that they explicitly state that they do not view DSE as a static problem, and thus designed their approach to be able to start from a specific existing configuration, enabling dynamic reconfiguration.

3.3 Summary

Surveying the state-of-the-art, we realized that there are two interrelated problems. First, there is a language design challenge in that it is difficult to describe what is modifiable about a system design in a way that is both complete and straightforward. Some of the most powerful languages for describing system design options (e.g., Kerzhner [29] or GuideArch [19]) were also the least abstract and potentially the most challenging to use by a standard system designer. Those that were easier to use were typically more focused on a single or small subset of all possible system changes. The notion of completeness is captured by the first row of Table 1 and the ease of use in the second row.

The second problem is the engineering challenge of creating an easy-to-use language and tool that minimizes additional effort necessary to incorporate it into a system design process. The creation of a custom model which is to be used exclusively for DSE and then discarded has a higher burden for incorporation into a system design process than one which re-uses existing system design artifacts. The best solutions we found were those that had tool support for modeling (see the third row of Table 1), often by re-using existing modeling languages (row four). Two existing approaches had strong support for domain-specific customization via extensible analyses (see row five) and three had the interactivity necessary to support design-by-shopping (row six).

4 Background

As discussed (see previous section), there are a number of tools that support design space exploration for system engineering. 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 custom-made fitness functions, or (c) worse still, do not support user-specified analyses at all. We decided to extend a well-established MBSE language (AADL, Section 4.1) and tool (OSATE, Section 4.2) with a configuration language that could support as many Degrees of Freedom as possible (see Section

```
1 system implementation Complete.PBA_spd_ctrl
2   subcomponents
3     spd_snsr: device snsr.spd;
4     throttle: device actuator.spd;
5     spd_ctrl: process ctrl.spd;
6     RT_1GHz: processor Real_Time.one_GHz;
7     std_marine_bus: bus Marine.std;
8     std_mem: memory RAM.std;
9   connections
10    DC1: port spd_snsr.snsr_dat -> spd_ctrl.snsr_dat;
11    DC2: port spd_ctrl.cmd -> throttle.cmd;
12    BAC1: bus access std_marine_bus <-> spd_snsr.BA1;
13    BAC2: bus access std_marine_bus <-> RT_1GHz.BA1;
14  properties
15    Allowed_Processor_Binding => (reference(RT_1GHz)) applies to
  ↪  spd_ctrl;
16    Allowed_Memory_Binding => (reference(std_mem)) applies to
  ↪  spd_ctrl;
17    Data_Rate => 5 KBytesps applies to DC1;
18 end Complete.PBA_spd_ctrl;
```

Listing 1: A simple system in AADL, adapted from [21]

4.3). That configuration language and tooling is then used to let designers "shop" for architectures (Section 4.4) using a well-established DSE tool (ATSV, Section 4.5). For the second phase of system exploration, we present a computationally expensive novel quantification strategy for system safety (Section 4.6).

4.1 Modeling Language: AADL

The *Architecture Analysis and Design Language* (AADL) is an internationally standardized architecture description language that was originally released in 2004 [21]. 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 those elements and connections. Elements and connections include both hardware and software, e.g., processors, processes, buses, memory, threads, subprograms, etc. [21] AADL specifies a number of standardized property sets, and system designers can also create custom properties from pre-existing or custom property types.

A simple AADL system is shown in Listing 1. It gives a taste for the AADL language (in its textual syntax), 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 [21]. The final section, lines 15-17, shows sample properties. AADL properties are typically used to guide system-level analyses that calculate, e.g., latencies, power consumption, weight totals, etc.

AADL is a declarative language, though most system analyses operate on what is known as the *instance model*. A declarative AADL model consists of component *type* declarations that define the interface of a component (i.e., its ports and other externally visible features) and component *implementation* declarations that define the internal structure of a component, namely its subcomponents and their interconnections. Component types and implementations are collectively referred to as *classifiers*. AADL supports extension of classifiers to add elements and refine the definitions of elements inherited from an extended classifier. The process of *instantiation* converts a declarative AADL model 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 (e.g., models of `process`es will be paired with the specific models of `processor`s they will run on in the final system) 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 [40], error propagations and transformations [42], and code generation targeting critical architectures [41].

4.2 Modeling Toolset: OSATE

There are a number of toolsets, both academic and commercial, that are designed to work with AADL (e.g., AADL Inspector[1], CAMET[2]). The Software Engineering Institute maintains the *Open Source Architecture Tool Environment*[3] (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.

4.3 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

[1] http://www.ellidiss.fr/public/wiki/inspector
[2] https://www.adventiumlabs.com/our-work/products-services/model-based-engineering-mbe-tools
[3] https://osate.org

obviously necessary, it is not immediately clear what should be changeable and what should be fixed when exploring a system's design space. Koziolek's work [31] contains two useful concepts for evaluating the expressiveness of a configuration language: an enumeration of several *Degrees of Freedom* (DoF, i.e., valid atomic system changes) and the six parameters required to completely describe those changes. Those parameters are:

1. **Changeable Elements:** Defined in part[4] as "The set of changeable metamodel elements" [31], these are the components, connections, etc. that can be modified.
2. **Primary Changeable Element:** Defined in part as "The primary changeable metamodel element" [31], this is the representative element of a set of model changes.
3. **Selection Rules:** Defined in part as "Rules to select the model elements to change" [31], these rules dictate what elements change along with a given primary changeable element.
4. **Value Rules:** Defined as "Rules to define the values that the selected model elements can take" [31], this can be thought of as type definitions for changeable elements.
5. **Interaction Constraints:** Defined in part as "A set of... constraints that may be violated by the selection and value rules because of interactions with other changes" [31], these constraints dictate requirements one changeable element places on the values of other changeable elements.
6. **Added Elements:** Defined in part as "A list of... elements this change type may add instances of" [31], these elements can be added (or, in reverse, removed) from a model rather than just being modifications of existing elements.

Collectively, these six pieces of information describe the Degrees of Freedom, which represent possible changes in a system architecture model, e.g., component selection, configuration parameter setting, allocation of software to hardware, etc. Koziolek focused on changes that affect performance, cost, and reliability, but notes that other quality attributes—such as security—could require other degrees of freedom.

[4] Though we attempt to adhere to the intuitive meaning behind Koziolek's definitions, we do not reproduce them in their entirety as they rely heavily on her specific approach and formalization. We elide things like variable or function names; refer to Table 6.3 of her dissertation for the full definitions [31].

4.4 Design By Shopping

Balling argues [5] that optimization techniques which require users to know and be able to quantify all of their preferences before seeing any candidate options (what Hwang and Masud term "a priori articulation of preference" [27]) are prohibitively difficult to use. The solution he suggests, which he terms *Design by Shopping*, 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 particular domains [5].

4.5 Design Space Exploration: ATSV

Penn State developed the *ARL Trade Space Visualizer* (ATSV) as an "engineering decision making tool" to support Design by Shopping [50, 49]. 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 quantifiable system characteristics in various graphical formats: glyph plots, histograms, parallel coordinates, scatter matrices, etc. [49] 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 inputs and measured values) are displayed.

ATSV 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 [48]. Then, as ATSV repeatedly queries the system model, it discerns which inputs affect which outputs using evolutionary algorithms. Other expected functionality is present as well, such as determining pareto optimality (i.e., designs where no preferred variable can be improved without worsening another preferred variable) [49] and hiding candidates that are infeasible according to user-specified constraints [44].

4.6 Subjective Assessment of Component Perfection

As mentioned previously (Section 2.2), one challenge with Design Space Exploration is the need to use exclusively quantitative analyses of a system. One possible

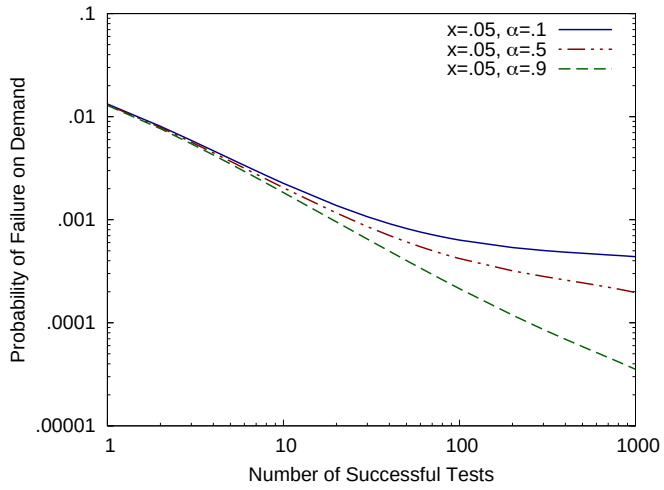

Fig. 1 Given a component with an estimated probability of failure on demand (*pfd*) of .0005, the effects of various estimates that the system is "perfect" (α) as the number of successful tests increase.

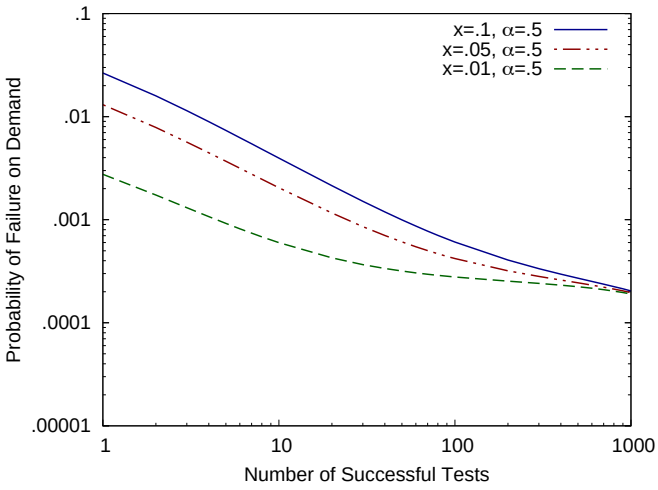

Fig. 2 Given a component with an estimated probability of failure on demand (*pfd*) of .0005, the effects of various levels of doubt (x) about the claimed *pfd* as the number of successful tests increase.

quantification strategy for reliability, which is closely related to safety, comes from the work of Bishop et al. It assumes the existence of a subjective assessor who is willing to provide an estimate of the likelihood that a component is "perfect," i.e., it will execute correctly for every possible demand[5] made of it [9]. This willingness to admit the possibility of component perfection has a significant, beneficial impact on a component's *probability of failure on demand* (*pfd*), i.e., the likelihood that a component will fail to produce a valid output when given a valid input. A component's estimated pfd can be refined further through testing; depending on the costs associated with those tests, an estimate that a component is perfect may be the difference between a feasible and infeasible testing burden.

Consider a hypothetical component with an estimated pfd of .0005. A similarly hypothetical assessor might estimate that the component is 50% likely to be perfect ($\alpha = .5$), and be only 90% certain that the estimated pfd is correct (i.e., have 10% doubt in the estimate; $x = .1$). Using Bishop et al.'s formula 12 [9], a conservative estimate of the component's pfd would be .1 before any tests have been performed; this is unacceptably high for many applications. The pfd drops rapidly, though, and after 137 successful tests, the conservative pfd is lower than the original estimate. More testing can be performed to continue to refine the estimate; Figures 1 and 2 show worst-case pfds of the component under different αs and xs.

[5] Notably, the definition of "demand" is somewhat flexible: Bertolino and Strigini define it as a sequence of inputs and suggest in the avionics domain an entire mission could be considered one demand [8].

Calculating these conservative pfds requires function optimization which may be too computationally expensive to perform on large numbers of system components. However, once the initial search space has been narrowed, an expensive analysis based on Bishop et al.'s work may prove quite valuable in differentiating between otherwise similar components. Our goal in using these methods is not to defend their validity for a specific domain or component, but rather to show how expensive calculations can be used as part of a multi-phased exploration of a system's design space.

5 The GATSE Process

GATSE's tooling consists of a collection of modifications to OSATE that support our configuration language, enable headless system instantiation and analysis, and 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 activities: trade space specification (Section 5.2) and trade space exploration (Section 5.3). We then discuss potential next steps in the system design process in Section 5.4.

5.1 Usage

The GATSE tooling is used through the OSATE and ATSV interfaces; there is no standalone GATSE exe-

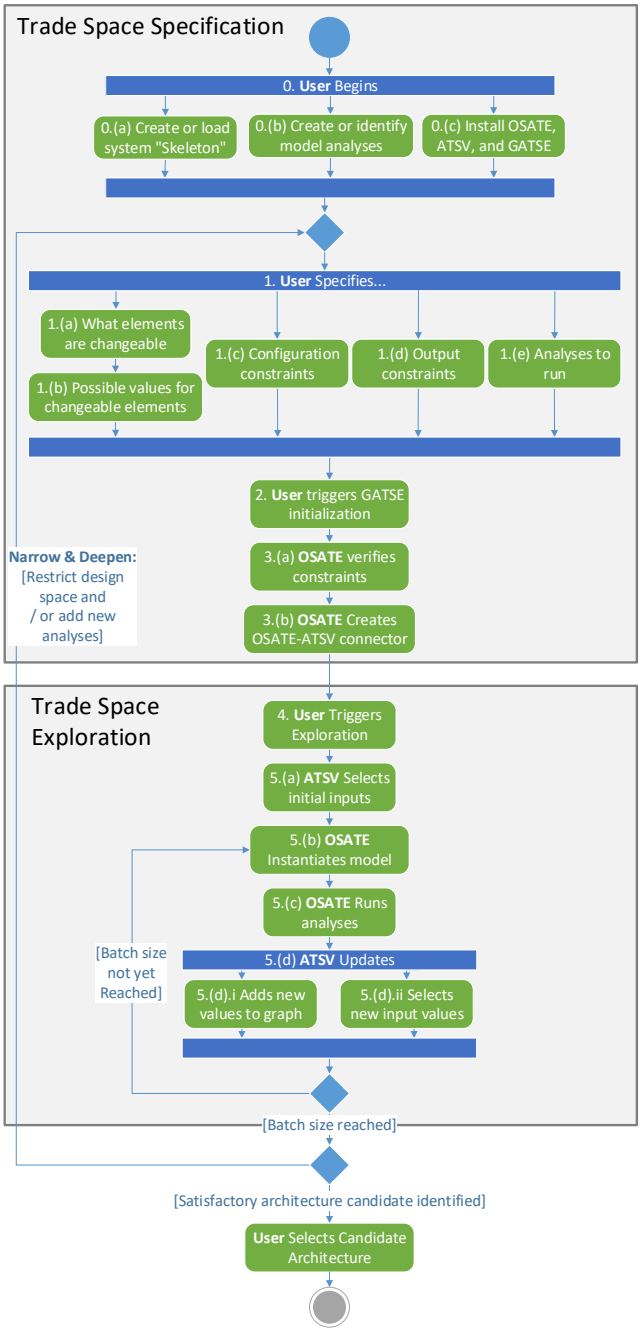

Fig. 3 An activity diagram showing the steps of using GATSE

cutable. Rather, an open-source[6] plugin is installed into OSATE which contains the necessary modifications as well as an installer for the connector and parser for ATSV. Use of the tool can be divided into two phases, *Trade Space Specification* (Steps 0-3) and *Trade Space Exploration* (Steps 4-5), with the user deciding to possibly repeat the process in Step 6. A typical use-case, as shown in Figure 3, is:

[6] https://github.com/osate/osate2-gtse

0. **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.
1. **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),
   (d) Output constraints (on model validity), and
   (e) Which analyses to run.
2. **User** Triggers GATSE initialization.
3. **OSATE** Initializes GATSE tooling:
   (a) Verifies element constraints are satisfiable, and
   (b) Creates OSATE-ATSV connection artifacts.
4. **User** Initializes ATSV:
   (a) Configures search parameters
   (b) Triggers design-space exploration
5. **GATSE** Explores the system's design space:
   (a) **ATSV** Selects initial 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 5b) until batch size is reached.
6. **User** Repeats (restricts design space and adds new analyses, i.e., returns to 1) until a satisfactory architecture is identified.

5.2 Trade Space Specification

The use of the GATSE tooling to specify a system's trade space mostly revolves around writing the system model's *configuration*: the file that indicates which model elements are changeable, what their possible values are, the analyses to run, which output variables to measure, and constraints on those outputs. The language that this file is written in is the topic of Section 6, here we discuss its use by the tooling.

5.2.1 Writing the Configuration

One of the features of the GATSE plugin to OSATE is a configuration editor that supports the language described in Section 6. In addition to the standard development-environment functionality, e.g., syntax highlighting, autocompletion, etc., we also have implemented static validation for the language; a screenshot of the validator is

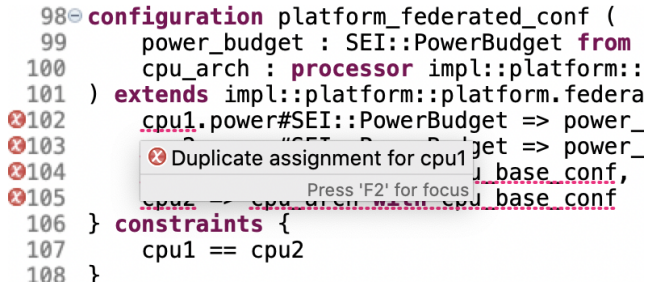

Fig. 4 The configuration language validator displaying an error in the file

shown in Figure 4. This validation checks for common issues like attempting to bind a property that does not apply to a given component type, duplicate configuration rules, cycles in the specification or potential classifier hierarchy, etc.

5.2.2 Processing the Configuration

Once the prerequisites (i.e., the three parts of Step 0 in Section 5.1) 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.

GATSE first verifies that the configuration is not over-constrained, i.e., the user's configuration constraints do not eliminate all possible system designs. Consider, for example, a system where both the processor and memory place requirements on the system's motherboard: these requirements may conflict and no satisfactory boards are available. Other over-constraining specifications may be far more complex. The verification 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 which uses variables with infinite types means that we lose soundness. However, the impact of this loss is not large, only that it may be possible to overconstrain a system (i.e., create a specification which does not allow any system candidates) when using unbounded integer- or real-typed properties and constraints.

We then remove constants from the equality logic using Kroening and Strichman's algorithm [32] and transform the equality logic to propositional logic using Zantema and Groote's equality substitution algorithm[7] [54]. The propositional logic is then transformed into conjunctive normal form using Tseitin's transformation [51] and satisfiability is checked using Sat4J [7].

If the configuration is satisfiable, i.e., one or more system architectures can be generated that meet all of the constraints, GATSE writes the necessary configuration and auxiliary files to support ATSV-OSATE integration. If the configuration is over-constrained, the GATSE initialization process is halted and the user is informed that they must remove one or more constraints.

5.2.3 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. These plugins need not be sophisticated and may only need a few lines of Java code. However, since analyses have full access to the system model, sophisticated analyses can be implemented as well.

5.3 Trade Space Exploration

Once the model has been built and the configuration specification has been processed, the actual design space exploration can begin, and it is here that all elements of GATSE work together. ATSV and OSATE will work together to rapidly (i.e., in less than a second) specify a concrete system candidate, instantiate it, run the selected battery of system analyses, and plot the results of those analyses in ATSV's graphical interface.

The user first optionally configures ATSV's search strategy, e.g., random search, maximizing / minimizing certain system attributes or analysis results, etc. Conflicts between desired attributes, such as high performance with low cost, will potentially result in multiple pareto-optimal candidates that will need to be explored further. Once satisfied with the search parameters, the user is ready to trigger the search (Step 4).

Next, it is time to begin instantiating system candidates. Recall from Section 4.1 that one purpose of instantiating AADL is determine which software elements (e.g., processes, port connections) will be allocated to which hardware elements (respectively: processors, buses) in the final system. In addition to the configuration-processing functionality described in Section 5.2.2, the GATSE plugin for OSATE also modifies

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

OSATE's instantiation logic. 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 second phase of instantiation, any properties customized by the configuration use their ATSV-specified value instead of the values specified in the skeleton model.

Once the model is fully instantiated, the user-specified analyses are executed. The results are returned to ATSV, which updates both the graphical display and its internal, evolutionary model with the new values. Unless this run was the end of a batch (the size of which is specified by the user), ATSV selects new values and continues execution, i.e., the process returns to Step 5.(b).

5.4 Next Steps

Once the design space has been explored, the user can either refine their search or select a candidate architecture.

5.4.1 Narrowing and Deepening the Search

If one of the resulting candidate architectures is satisfactory, the use of GATSE is complete. If multiple feasible candidate architectures remain, however, the configuration can be modified to continue the search, i.e., the process returns to step 1. The search space can be narrowed, by eliminating infeasible elements or element values, and / or deepened, by adding additional—likely more computationally expensive—analyses. These steps are taken by the system designer in response to the tool's outputs; it is not an automated step. Rather, this is equivalent to a shopper evaluating a number of candidate items and then refining their search to learn more information about a subset of the candidates. While the particular steps taken by the designer will typically be specific to the system itself, they will almost certainly involve reducing the search space (by specifying fewer options in Step 1(a)-1(b)), applying additional output constraints (Step 1(d)), and selecting additional analyses to run (Step 1(e)).

Note that this step alleviates, to some extent, the second challenge discussed in Section 2.2: the need for rapid performance. Recall that there is, in general, a tradeoff between the depth of analyses performed and the number of candidates that can be analyzed. Rather than focus on the performance of our particular tool implementation, GATSE enables a more general approach: an iterative, multi-phase process of tradespace exploration. Initial exploration can be conducted using a minimal set of system analyses that can be executed relatively quickly. Then, based on the results of this initial exploration, the tradespace can be winnowed down. The resulting subset can be explored using more (and more computationally expensive) analyses that provide richer information on individual system models.

5.4.2 Selecting a Candidate Architecture

Once the number of candidate architectures has been reduced to a manageable number, traditional MBSE processes (e.g., model review by a human system architect, negotiation with stakeholders, etc.) will likely be necessary for selecting a final design. These processes may involve metrics for which no automated or quantitative analysis exists, e.g., subjective measures of safety, security, feasibility of construction, familiarity with certain components or their manufacturers, etc. If all relevant system metrics can be quantified, though, then determining the final system configuration should be a straightforward choice between any of the pareto-optimal candidate architectures identified by ATSV. This choice can be made by any appropriate criterion. With the system components and their configurations completely specified by GATSE, construction of the system should be straightforward.

6 The Choicepoint Language

In this section we describe the configuration language used in step one of the process defined in Section 5.1 and Figure 3. The language, is implemented[8] in XText[9], but we provide a grammar in EBNF in Listing 2.

6.1 Walkthrough

An example configuration file, corresponding to the system from Listing 1 and containing only one configuration specification, is shown in Listing 3. Lines 3 and 4 are the specification's `parameters` (see Section 6.1.1), lines 5-7 are the `extends` clause (see Section 6.1.2), and lines 8-10 are `constraints` (see Section 6.1.3). 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`.

[8] See the `org.osate.gtse.config.*` packages in https://github.com/osate/osate2-gtse

[9] https://www.eclipse.org/Xtext/

```
Configuration=Root,Configurations,Analyses,Outputs;
Root='root',ID;
Configurations=Config,{Config};
Config='configuration',ID,[Params],
  ↪  ['extends',ID,[With]],[Assignments],[Constraints];
With='with',Combination,{'&',Combination};
Combination=ID,[Args];
Params='(',[ConfParam,{',',ConfParam}],')';
ConfParam=ID,':',(FClassifierType|FPropertyType),
  ↪  [Candidates];
FClassifierType=?ComponentCategory?,CNAME;
FPropertyType=PNAME;
Candidates='from','(',[Candidate,{',',Candidate}],')';
Candidate=CNAME|?PropertyExpr?;
Assignments='{',[Assig,{',',Assig]},'}';
Assig=LVal,'=>',ConfVal;
LVal=(('*',|,ElemRef),['#',PNAME])|'#',PNAME;
ConfVal=CNAME,[Args],[With],[Assigs]|?PropertyExpr?|
  ↪  Assigs;
Args='(',[Arg,{',',Arg}],')';
Arg=ID,'=>',ConfVal;
ElemRef=ID,{'.',ID};
Constraints='constraints','{',[Constr,{',',Constr]},'}';
Constr=Cond,[Relation,Cond];
Cond=CondExpr,Relation,CondExpr;
CondExpr=ConfElem|CondVal|SetVal;
SetVal='{',CondVal,{',',CondVal},'}';
CondVal='!',CNAME|?PropertyExpr?;
ConfElem=(ElemRef,['#',PNAME])|'#',PNAME;
Analyses='analyses','{',String,{String},'}';
Outputs='outputs','{',Variable,{',',Variable},'}';
Variable=ID,':',[Type],[Limit];
Type='int'|'float'|'string';
Limit=Relation,?IntegerTerm?|?RealTerm?|?StringTerm?;
Relation='>'|'>='|'=='|'!='|'<'|'<='|'forbids'
  ↪  |'requires'|'in';
(* AADL classifier or configuration name *)
CNAME=ID {'::' ID} ['.' ID];
(* AADL property name *)
PNAME=ID ['::' ID]
```

Listing 2: The configuration language grammar in EBNF. AADL constructs are denoted with ?s.

6.1.1 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 3 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.

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

Listing 3: A configuration for the system from Figure 1

6.1.2 Extends

A configuration's `extends` clause specifies which elements of the skeleton model the configuration applies to (line 5 in Listing 3) 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[10] 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, this can be done using a `with` statement (not shown in Listing 3). `with` statements are a more general purpose construct, though, that lets designers combine multiple configuration specifications for the same element.

6.1.3 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 3: the lower quality and less expensive sensors cannot support higher data rates (lines 9-10). Six types of constraints are supported; Table 2 gives their syntax and informal semantics.

[10] Qualified relative to the extended element, see line 5 of Listing 3

Syntax	Semantics
A==B	Elements A and B must be the same
A!=B	Elements A and B must not be the same
A==X requires B == Y	If element A is X, B must be Y
A==X forbids B == Y	If element A is X, B must not be Y
A==X requires B in {Y, Z}	If element A is X, B must be Y or Z
A==X forbids B in {Y, Z}	If element A is X, B must not be Y or Z

Table 2 Constraint syntax and semantics.

7 Use Case: A Wheel Brake System

We used a model of a fictional aircraft wheel brake system (WBS) as a case study. Our objectives are twofold. First, we use the case study to illustrate the GATSE tool and process. Second, the study allows us to evaluate the expressiveness and ease of use of our configuration language, the difficulty of adding custom analyses, and evaluate the tool's performance. This system is fairly well-studied in the critical-system and model-based engineering literature: it was originally created as part of the ARP4754 [45] and ARP4761 [46] standards, and has been described and (re)modeled as part of a number of efforts since then [10, 47].

The WBS 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 [53]. As more hardware components and software modules become available, it becomes increasingly challenging to understand which combination of them is best, as there are a large set of desired—and to some extent competing—quality attributes such as cost, power consumption, latency / performance, weight, and efficacy.

The system model, GATSE configuration files, generated results, and both custom analyses described in this section are open-source and publicly available[11].

[11] https://github.com/osate/osate2-gtse/

7.1 System Description

A simplified view of the WBS architecture is shown using AADL's graphical notation in Figure 5; 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 [39, 47]:

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. An expanded view, showing the platform, subsystem, and selector subcomponents, is shown in Figure 7.
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:** An 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, status, alert, electrical, and steering) which we do not elaborate, but include as "black box" components without implementations, see Figure 6. While these could be refined later if necessary, by annotating them with expected failure rates and error propagations, we can reason about how different component options impact the system's overall safety. That is, if we have, e.g., three different steering subassemblies, we can include them in our tradespace exploration—even without knowing their implementations—by having three different black-box components with different safety characteristics specified.

7.2 Trade Space Specification

For the purposes of testing, we created two or three options for several WBS classifiers (e.g., component types,

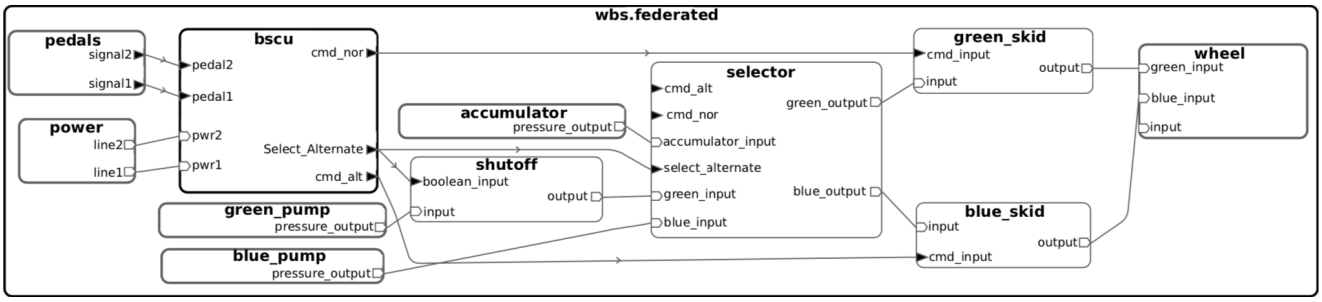

Fig. 5 An implementation view of the wheel braking system's (WBS) architecture. Adapted from Delange et al.[14]

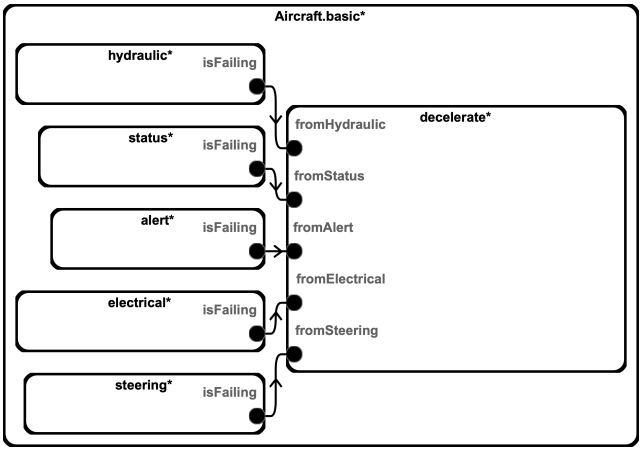

Fig. 6 A functional view of several WBS subsystems. These are treated as "black boxes," without implementation details, but failure rates and error propagation information are specified.

Name	Options	Red?	Const?	Type?
Subcomponent Choices				
`Monitor` software	2	Y	Y	N
CPU Architecture	3	Y	N	Y
CPU Power Supply	2	Y	N	Y
Hydraulic Pump	2	Y	N	N
Steering assembly	3	N	N	N
Component Interconnections				
Power bus	6	Y	N	Y
Property Specifications				
CPU Power	∞	Y	N	N

Table 3 Selected choicepoints in the WBS. *Red:* is the element is part of a redundant assembly? *Const:* is the choice constrained? *Type:* are both the type and implementation variable?

implementations, hardware interconnects), and affixed new properties. Examples of these are given in Table 3, and a small section of the full configuration file is shown in Listing 4. The number of possible configurations grows rapidly: even setting aside the CPU Power property, there are $2 \times 3 \times 2 \times 2 \times 3 \times 6 = 432$ possible configurations using only the choices from Table 3. Even in our relatively small example system, a brute

```
configuration platform_federated_conf (
  power_budget : SEI::PowerBudget from (0.1W .. 300W),
  cpu_arch : processor impl::platform::cpu from
    ↪  (impl::platform::cpu.x86, impl::platform::cpu.x64,
    ↪  impl::platform::cpu.arm)
) extends impl::platform::platform.federated {
  cpu1 => cpu_arch with cpu_base_conf {
    power#SEI::PowerBudget => power_budget
  },
  cpu2 => cpu_arch with cpu_base_conf {
    power#SEI::PowerBudget => power_budget
  }
} constraints {
  cpu1 == cpu2
}
```

Listing 4: A snippet of the full WBS configuration, specifying choices and constraints for the CPUs used in the federated architecture.

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.
3. **Identical CPU Architectures within Assemblies:** In a federated architecture, we required that the CPUs used the same architecture.

We enabled a relatively small number of system analyses for our initial searches. Our expectation is that the number and analytical 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 func-

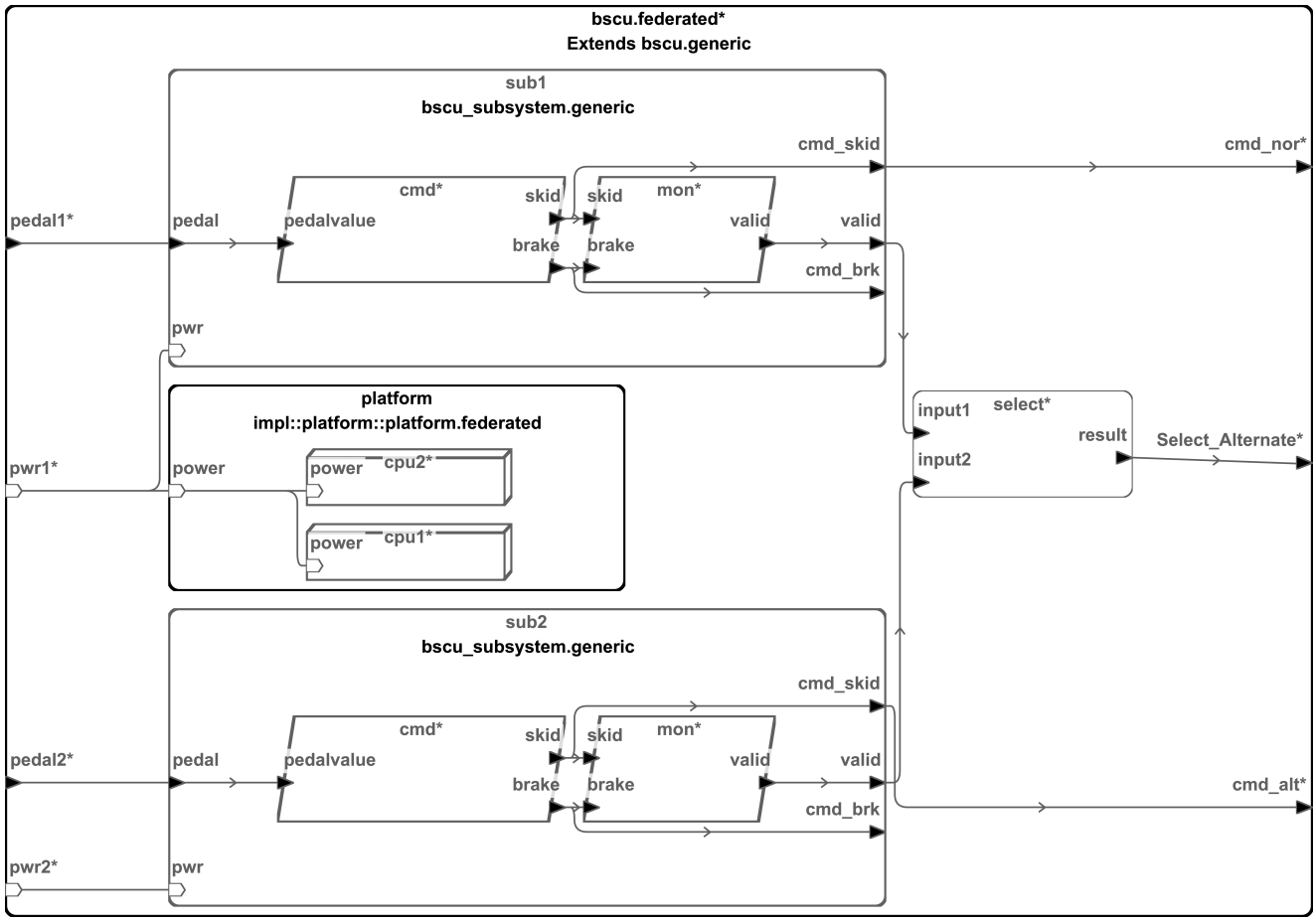

Fig. 7 The implementation of the WBS's Brake System Control Unit (BSCU)

tion 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 8 is the complete source code of the braking power analysis. It relies on property annotations like those in the top-left portion, including built-in properties from standard Software Engineering Institute (SEI) provided property sets, as well as a custom property set for this use case / demonstration.

7.3 Trade Space Exploration

The goal of tooling used in Design by Shopping is to make clear to the system architect the correlations and tradeoffs between a system's various quality attributes. In order to achieve this, system information should be presented in an intuitive, graphical tool and used to refine a system's design parameters and ultimately select a candidate system architecture [5].

The three boxes in the lower half of Figure 8 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 let designers view the system design trade space in a number of formats, e.g., the scatter plot in the lower left of Figure 8 as well as parallel coordinate and histogram plots (these are not particularly helpful in the WBS use case, and are 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 val-

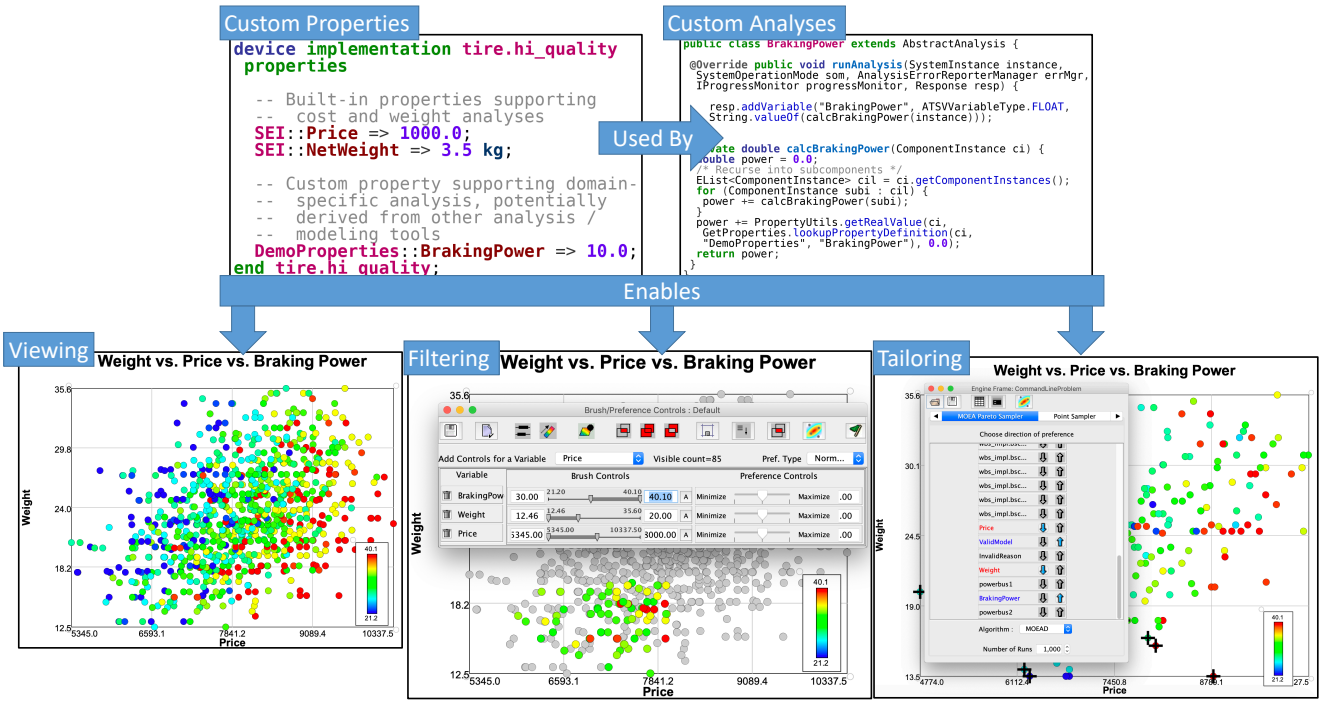

Fig. 8 The motivating vision behind the GATSE project. The top left shows model properties, which are used by the analysis in the top right. The bottom row shows different views of the WBS system's trade space in ATSV [50].

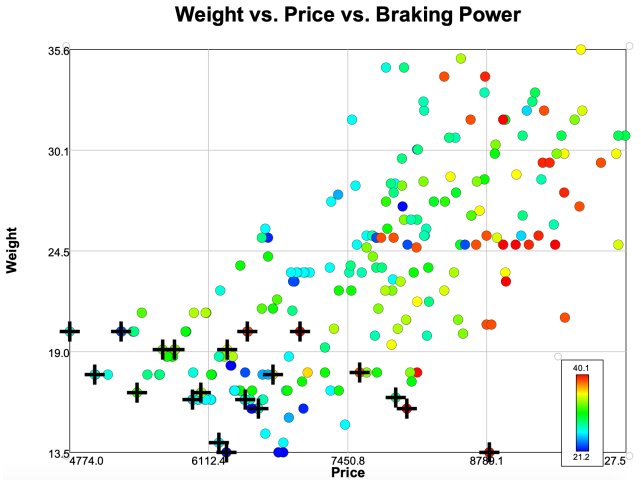

Fig. 9 Candidate WBS architectures viewed in ATSV [49]. The color of each point represents the system's *braking power*, our stand-in for a domain specific analysis. The points marked with a + are pareto optimal.

ues 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 3), it is also possible to filter the views directly in ATSV, as in the lower middle portion of Figure 8.
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 as it lets us maximize or minimize any number of system variables or analysis results (e.g., both a specific component's individual property setting and / or an overall measured system value can be optimized). It can be configured using the interface shown in the lower right of Figure 8. Pareto optimal designs can also be highlighted, these are identified automatically by ATSV when using a Pareto Sampler. Figure 9 shows the WBS trade space with pareto-optimal architectures marked.

7.4 Next Steps

Now that the WBS's trade space has been defined, we can refine the search space and eventually compare pareto-optimal system architectures.

7.4.1 Narrowing and Deepening

After surveying the initial trade space, a designer may want to focus their search on fewer candidate architec-

tures, but examine them deeply with more computationally expensive analyses. After exploring the trade space, it is straightforward to identify options that are rarely or never chosen in candidates that performed well for important characteristics. Recall that some of the characteristics we initially focused on in the WBS were price, weight, and our stand-in for a domain specific analysis, "braking power." We discovered that inexpensive, lightweight, and powerful braking systems typically used system power (rather than standalone backups), higher-quality tires, and relatively powerful hydraulic pumps. With this narrower search space, a user can afford to allocate more time to analyzing each candidate architecture; this is done by enabling additional analyses in the configuration file.

We created a second custom analysis that incorporates the probability of failure on demand (pfd) calculations from Section 4.6. We modified an existing OSATE plugin that generates fault trees (a standard way of assessing a system's safety [18]) from annotated AADL models [20] to use new properties which describe the subjective assessor's beliefs about a component and the number of demands it has successfully passed. We then extended the WBS model with example values (i.e., assessments of a component's estimated pfd, likelihood of perfection, and doubt about the claimed pfd) and used them to calculate the overall failure rate of the WBS. That overall value was then exposed to ATSV, and thus can be used for viewing, filtering, and tailoring as described in Section 7.3. These modifications were fairly straightforward to implement, and extending GATSE to include our calculations was far simpler than correctly implementing the pfd calculation itself.

7.4.2 Selecting a Candidate Architecture

If the number of candidate architectures generated is now small enough that some merit close inspection, ATSV supports a fourth activity:

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 10 shows an example of one candidate architecture for the WBS.

At this point in our use case, system designers would likely need to look beyond the quantitative analyses available in GATSE to compare the pareto optimal candidate architectures. This process is out of scope for this publication, but would proceed according to the requirements of the organization contracted to produce the WBS.

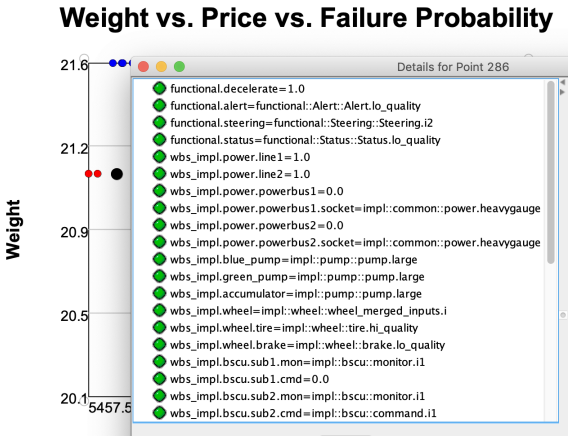

Fig. 10 Detailed system information generated / calculated by GATSE for a WBS candidate architecture. This will be displayed when a point in a plot in ATSV is selected [49]

8 Evaluation

Recall from Section 2 that we had six high-level objectives derived from our review of the state-of-the-art. By using OSATE and AADL, we met objectives 3 and 4 (tool support for modeling and a re-usable modeling language). By using ATSV, we met objective 6 (support for Design-by-Shopping). We now discuss the degree to which the other objectives were met, and also comment on the performance of our approach.

8.1 Expressive Configuration Language

We took as our goal to support as much of Koziolek's Degrees of Freedom [31] as possible. In that work, system changes are specified using six parameters; GATSE supports (at least partially) the first five:

1. **Changeable Elements:** Full support. We support changesets of size one directly, and larger sizes through constraints. That is, instead of changing elements A, B, and C simultaneously, a GATSE user must change A and then use a constraint to require particular values of B and C given the particular value of A.
2. **Primary Changeable Element:** Full support. This is trivially supported by our implementation as our changesets can only have a size of one.
3. **Selection Rules:** Partial support. We claim only partial support since some aspects of an AADL model

are not addressable with our current implementation. As component types, implementations, and properties make up the bulk of AADL specifications, GATSE does not allow changing any annexes, flows, and some other elements directly. In practice, however, doing so was not necessary and we did not find this to be a significant limitation since components and properties containing those annexes, flows, etc. are addressable and can be directly changed. If it were to be necessary, we are confident the language could be extended to address these additional elements.

4. **Value Rules:** Full support. These are encoded in the `from` clause of the parameter of a choicepoint specification.
5. **Interaction Constraints:** Partial support. We claim only partial support because some interactions (e.g., those involving relations other than equality, inequality, and tests of set membership) cannot be specified in the current implementation. We note that this is a restriction of both ATSV and the way satisfiability is checked in GATSE. We further note that these constraints are only necessary in dynamic DSE tooling, i.e., tools where the search can be guided using a fitness function. Static tools (such as AFFMAD [2]) must first be extended to support dynamic exploration before support for constraints would be usable.
6. **Added Elements:** No support. We cannot claim any support since it is impossible to add (or remove) elements from an architecture using our current implementation.

We are satisfied with the capabilities of our language (and associated tooling) insofar as it enables the specification and dynamic exploration of a system's design space, supports easily extensible analyses, and supports most of the information required by Koziolek's Degrees of Freedom. It compares favorably with existing work, most of which is more narrowly focused than our approach.

8.2 Easily-Used Configuration Language

The configuration language we created for this work is powerful, but can be challenging to use. That is, the time and / or training necessary to become proficient is likely to be non-trivial. Much of this difficulty stems from the complexity of the problem it addresses, which is to succinctly describe a potentially broad set of modifications to a system, which may rely on or impact other modifications, which may themselves rely on or impact yet other modifications, etc. Note that this challenge is closely related to the previous objective: as a configuration language is extended to support more types of changes, its complexity will necessarily increase. That is, while clever language design and tooling extensions can provide a great deal of aid to system designers, to some extent these objectives are in competition and improving at one will degrade the other.

Compared to the approaches discussed in Section 3, our language is roughly as powerful as the state-of-the-art, and, by aligning with a standardized modeling language, should be usable by system designers who are generally familiar with MBSE. Specific syntactical issues can still be challenging to fix, though the validator we have built for the language is helpful here. Ultimately, we cannot call ourselves entirely successful in this objective; more work will be needed to ensure system configurations are relatively straightforward to write.

8.3 Support for Extensible Analyses

We found the process of extending GATSE with a custom, domain specific analysis to be fairly straightforward. GATSE's extensible, plugin-based architecture enables analysis developers to easily traverse an instantiated system model to, e.g., read in relevant property values. These values can then be used in any quantitative analysis of the system.

While the initial version of this work [36] had only the fairly simple Braking Power analysis, in this paper we have described the more heavyweight failure probability analysis (see Section 4.6). This analysis calculates a system's probability of failure on demand by repeatedly optimizing a nonlinear function, which requires fairly sophisticated statistical software that is easily available via the Apache Common's Math3 library[12]. Adding this analysis to GATSE was simple: we added custom properties to our model, extended a single interface so we could read in those properties, and then passed their values to a class that implements the pfd calculations. The results of those calculations were put into the key-value store that OSATE returns to ATSV, and from there were both plotted visually and used to guide the search for new candidate architectures.

8.4 Performance

Though performance was not considered a high priority in the development of GATSE's initial prototypes, we

[12] https://commons.apache.org/proper/commons-math/

	Step 5.(d)	Step 5.(b)		Step 5.(c)		
Time (seconds)	ATSV	Unmarshalling	Instantiation	Analysis	Marshalling	**Total**
Initial Search	.1159	.1213	.1750	.1683	.0002	**.5807**
Deeper Search	.1173	.1210	.1794	.2121	.0002	**.6301**

Table 4 Average duration, over 1,000 executions, of various steps performed by ATSV and OSATE. Other OSATE initialization tasks, loading analysis plugins, and transmitting the response over the loopback interface typically took less than a thousandth of a second. Data from the first execution was removed from the analysis in order to obtain GATSE's steady-state performance. Steps refer to the process from Section 5.1 / Figure 3.

noted previously (see Section 2.2) that the tool's usability would be improved by decreasing the time it takes to identify and analyze a given number of candidate architectures. Table 4 shows detailed performance data obtained from the WBS use case. Identifying, instantiating, and analyzing a single system architecture takes roughly .58 seconds (or .63 seconds with the slower safety analysis); generating the initial thousand architectures used for Figure 8 took roughly 9 minutes and 59 seconds[13] (generating 300 architectures with the safety analysis took roughly 3 minutes and 13 seconds). Interestingly, even with the more heavyweight analysis overall execution time is not dominated by any particular factor: analysis and model instantiation make up the bulk of the execution time. We had expected a larger portion of the overall execution time to come from model analysis, we are encouraged that performance was relatively good. That said, overall optimizations to OSATE, in particular its instantiation logic (which takes roughly .18s, see Table 4), are more important than we had realized. We explored the tradespace of the WBS on a machine with an Intel i9 CPU running at 2.4GHz and 32 gigabytes of memory. We used OSATE 2.9.1, GATSE 1.0.0.202012042351, and ATSV 10.0.8.32bit.

8.5 Limitations of GATSE

In summary, we were successful in marrying a standardized MBSE language, existing MBSE tool, and standalone DSE software. This is significant in that it supports a MBSE and DSE process that is broadly usable in the critical system space. However, while we have met a number of our objectives, we need more data on some and others will require continued work. Specifically, the limitations we see of the approach are:

1. **Missing support for adding/removing components:** Our configuration language and tooling cannot describe this parameter from Koziolek's DoF framework. This limits the types of system changes describable by our tool.
2. **A challenging configuration language:** The complexity of our configuration language may be endemic to the domain, but we are not satisfied that it is as easy-to-use as possible
3. **A lack of industrial-use data:** Our use case was helpful in illuminating a number of aspects of the tool, but it is no substitute for getting the tool in the hands of practitioners and soliciting their feedback.
4. **Limited soundness checking for constraints:** Because our constraint checking, as described in Section 5.2.2, only operates on a subset of property types usable in AADL, it lacks soundness.

[13] We note that this is roughly five times faster than the initial reported speed [36]. The speedup is primarily due to improved hardware.

9 Future Work

We plan on addressing the limitations identified in the previous section by working with the AADL Standardization Committee, industrial users of OSATE, and continuing to refine our software.

9.1 Language: Simplify and Integrate into AADL

As we have based our work on AADL, we have a significant opportunity to standardize system configuration specification techniques in future revisions of the AADL standard. What's more, there are other uses for these specifications, and a single, standardized approach would lead to more re-use of modeling artifacts, a smoother learning curve, and include the ability to address all AADL elements, such as annexes and flows. There are important lessons we can draw on from the literature, e.g., aspect orientation [30], which enables a strong separation of concerns, or a proposed metamodel for design space exploration [15] with which we could align. The configuration language developed for this work has already seeded conversations and prompted refinements within the AADL standard committee; we will continue to pursue this line of improvement by evaluating the incorporation of aspect orientation and the

proposed metamodel. In these conversations and revisions, we are particularly focused on simplifying the language so that it is easy to use while remaining sufficiently expressive.

9.2 Evaluation: Work with Industry

We presented an early version of this work to an audience with industrial users in late 2017[35]. There has been interest in GATSE since that time, though until recently we were focused on maturing the language and tooling. As we are now satisfied with its maturity, we plan on working with these industrial partners to perform an larger evaluation of GATSE. In particular, we are interested in system designer's feedback on the usability of the tool, and its applicability to real-world design issues.

9.3 Tooling: Soundness Checking for Constraints

We aim to improve the GATSE tooling by making our constraint checking sound. In order to do this, we will need to be able to check constraints placed on properties that have unbounded integer or real types, which we plan on doing by migrating from our SAT solver backend (Sat4J [7]) to a SMT solver such as Z3 [33], which will be able to operate on a much larger subset of AADL's property types. This would allow us to check, e.g., constraints such as "CPU1 Power Consumption must be less than 50.000W."

9.4 Tooling: Performance

OSATE was 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 as when used as part of GATSE. 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 [28]. While performance has improved—primarily due to improvements in CPU / memory speed—since the initial description of GATSE [36], the Software Engineering Institute continues to optimize OSATE itself as well. This work continues independently of GATSE, but the effects will benefit the performance of our tool.

10 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 trade 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 began this effort by aiming to understand the overlap and potential synergy between MBSE and DSE techniques. Based on the state-of-the-art, we laid out six objectives (see Section 2). Overall, the effort was a qualified (see Section 8) success. It was a success in that, we argue, GATSE represents an advancement over the state-of-art as it includes many of the features found in similar projects (see Section 3), and also enables design-by-shopping [5] using 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 (a standard workbench used to develop AADL specifications) there is significant opportunity for DSE models to be reused beyond GATSE. 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.

Acknowledgements

The authors wish to thank Julien Delange, Min Young Nam, and Peter Feiler for the original concept and feedback; Joseph Seibel for the configuration validator implementation; and the anonymous reviewers for their feedback which has been invaluable in improving this paper. 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.

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.

References

1. Abdeen, H., Nagy, A.S., Varró, D., Hegedüs, Á., Sahraoui, H., Horváth, Á.: Multi-objective optimization in rule-based design space exploration. In: ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 289–300. Association for Computing Machinery, Inc, Vasteras, Sweden (2014). DOI 10.1145/2642937.2643005. URL http://dl.acm.org/citation.cfm?doid=2642937.2643005
2. Adventium Labs: https://www.adventiumlabs.com/demonstration-combined-use-dse-rbd-and-tse-trade-space-analysis (2017). Accessed: August 15, 2018
3. Aleti, A., Bjornander, S., Grunske, L., Meedeniya, I.: ArcheOpterix: An extendable tool for architecture optimization of AADL models. In: 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pp. 61–71. IEEE, Vancouver, Canada (2009). DOI 10.1109/MOMPES.2009.5069138
4. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and meta-models in clafer: Mixed, specialized, and coupled. In: B. Malloy, S. Staab, M. van den Brand (eds.) Software Language Engineering (SLE10), pp. 102–122. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
5. Balling, R.: Design By Shopping: A New Paradigm? In: Proceedings of the Third World Congress of Structural and Multidisciplinary Optimization (WCSMO-3), pp. 295–297. Buffalo, NY, USA (1999)
6. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software **82**(1), 3–22 (2009). DOI 10.1016/J.JSS.2008.03.066
7. Berre, D.L., Parrain, A.: The SAT4J library, Release 2.2, System Description. Journal on Satisfiability, Boolean Modeling and Computation **7**, 59–64 (2010)
8. Bertolino, A., Strigini, L.: Assessing the risk due to software faults: estimates of failure rate versus evidence of perfection. Software Testing, Verification and Reliability **8**(3), 155–166 (1998). DOI 10.1002/(SICI)1099-1689(1998090)8:3<155::AID-STVR163>3.0.CO;2-B. URL http://onlinelibrary.wiley.com/doi/10.1002/(SICI)1099-1689(1998090)8:3{%}3C155::AID-STVR163{%}3E3.0.CO;2-B/full
9. Bishop, P., Bloomfield, R., Littlewood, B., Povyakalo, A., Wright, D.: Toward a Formalism for Conservative Claims about the Dependability of Software-Based Systems. IEEE Transactions on Software Engineering **37**(5), 708–717 (2011). DOI 10.1109/TSE.2010.67. URL http://ieeexplore.ieee.org/document/5492693/
10. Bozzano, M., Cimatti, A., Fernandes Pires, A., Jones, D., Kimberly, G., Petri, T., Robinson, R., Tonetta, S.: Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In: D. Kroening, C. Păsăreanu (eds.) Computer Aided Verification (CAV), pp. 518–535. Springer, Cham, San Francisco, California, USA (2015). DOI 10.1007/978-3-319-21690-4_36
11. Chilenski, J.J., Ward, D.T.: System Architecture Virtual Integration SAVI AFE 59S1 Report Summary Final Report. Tech. rep., System Architecture Virtual Integration (2015)
12. Clark, B., Miller, C., McCurley, J., Zubrow, D., Brown, R., Zuccher, M.: Department of Defense Software Factbook. Tech. Rep. CMU/SEI-2017-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2017)
13. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation **6**(2), 182–197 (2002). DOI 10.1109/4235.996017
14. Delange, J., Feiler, P., Gluch, D., Hudak, J.: AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment. Tech. rep., Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (2014)
15. Diewald, A., Voss, S., Barner, S.: A Lightweight Design Space Exploration and Optimization Language. In: Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems - SCOPES '16, pp. 190–193. ACM Press, New York, New York, USA (2016). DOI 10.1145/2906363.2906367
16. DoD Architecture Framework Working Group: DoD Architecture Framework Version 1.0. Tech. rep., United States Department of Defense (2003)
17. Eder, J., Voss, S.: Usable Design Space Exploration in AutoFOCUS3. In: Workshop on Open Source Software for Model-Driven Engineering (OSS4MDE), in conjunction with MODELS conference. Brittany, France (2016)
18. Ericson II, C.A.: Hazard Analysis Techniques for System Safety, second edn. John Wiley & Sons (2016)
19. Esfahani, N., Malek, S., Razavi, K.: GuideArch: guiding the exploration of architectural solution space under uncertainty. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 43–52. IEEE Press, San Francisco, USA (2013)
20. Feiler, P., Delange, J.: Automated fault tree analysis from aadl models. Ada Lett. **36**(2), 39–46 (2017). DOI 10.1145/3092893.3092900. URL https://doi.org/10.1145/3092893.3092900
21. Feiler, P., Gluch, D.: Model-Based Engineering with AADL, 1st edn. Addison-Wesley Professional, Upper Saddle River, NJ (2012)
22. Feiler, P., Hansson, J., de Niz, D., Wrage, L.: System Architecture Virtual Integration: An Industrial Case Study. Tech. rep., Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2009)
23. Frank, S., van Hoorn, A.: SQuAT-Vis: Visualization and Interaction in Software Architecture Optimization. In: A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, O. Zimmermann (eds.) European Conference on Software Architecture (ECSA20), pp. 107–119. Springer, Cham, L'Aquila, Italy (2020). DOI 10.1007/978-3-030-59155-7_9. URL https://doi.org/10.1007/978-3-030-59155-7{_}9
24. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language. Morgan Kaufmann (2014)
25. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, Techniques, and Applications.

ACM Computing Surveys **45**(1), 1–61 (2012). DOI 10.1145/2379776.2379787
26. Hegedüs, Á., Horváth, Á., Varró, D.: A model-driven framework for guided design space exploration. Automated Software Engineering **22**(3), 399–436 (2015). DOI 10.1007/s10515-014-0163-1. URL `https://link.springer.com/article/10.1007/s10515-014-0163-1`
27. Hwang, C.L., Masud, A.S.M.: Multiple Objective Decision Making – Methods and Applications: A State-of-the-Art Survey, *Lecture Notes in Economics and Mathematical Systems*, vol. 164. Springer-Verlag Heidelberg (1979)
28. Iacobucci, J.V.: Rapid Architecture Alternative Modeling (Raam): a Framework for Capability-Based Analysis of System of Systems Architectures. Ph.D. thesis, Georgia Institute of Technology (2012)
29. Kerzhner, A.A.: Using logic-based approaches to explore system architectures for systems engineering. Ph.D. thesis, Georgia Institute of Technology (2012)
30. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: M. Akşit, S. Matsuoka (eds.) ECOOP'97 — Object-Oriented Programming, pp. 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
31. Koziolek, A.: Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes. Ph.D. thesis, Institut für Programmstrukturen und Datenorganisation (IPD) (2013). DOI 10.5445/KSP/1000032342
32. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View, second edn. Springer-Verlag, Berlin, Germany (2016). DOI 10.1007/978-3-662-50497-0
33. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS08), pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
34. Owens, B., Leveson, N.: A comparative look at MBU hazard analysis techniques. In: Annual Military and Aerospace Programmable Logic Device International Conference(MAPLD), pp. 1–11. Washington DC, USA (2006). URL `http://sunnyday.mit.edu/papers/Owens-mapld.pdf`
35. Procter, S.: Guided architecture trade space exploration for safety-critical software systems. Presentation (2017)
36. Procter, S., Wrage, L.: Guided architecture trade space exploration: Fusing model based engineering design by shopping. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 117–127 (2019). DOI 10.1109/MODELS.2019.000-9
37. Rago, A., Vidal, S., Andres Diaz-Pace, J., Frank, S., Van Hoorn, A.: Distributed qality-atribute optimization of sofware architectures. In: Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS17), vol. Part F1306, pp. 1–10. Association for Computing Machinery, Fortaleza, CE, Brazil (2017). DOI 10.1145/3132498.3132509. URL `http://dl.acm.org/citation.cfm?doid=3132498.3132509`
38. Ross, J.A., Murashkin, A., Liang, J.H., Antkiewicz, M., Czarnecki, K.: Synthesis and exploration of multi-level, multi-perspective architectures of automotive embedded systems. Software & Systems Modeling pp. 1–29 (2017). DOI 10.1007/s10270-017-0592-y
39. SAE Aerospace: AIR6110: Contiguous Aircraft/System Development Process Example. Tech. rep., SAE International (2011)
40. SAE AS-2C Architecture Description Language Subcommittee: SAE Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Behavior Annex. Tech. rep., SAE International (2011)
41. SAE AS-2C Architecture Description Language Subcommittee: SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex. Tech. rep., SAE International (2015)
42. SAE AS-2C Architecture Description Language Subcommittee: SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex E: Error Model Annex. Tech. rep., SAE International (2015)
43. Selva, D., Crawley, E.F.: VASSAR: Value assessment of system architectures using rules. In: IEEE Aerospace Conference Proceedings, pp. 1–21. IEEE, Big Sky, Montana (2013). DOI 10.1109/AERO.2013.6496936
44. Simpson, T., Carlsen, D., Congdon, C., Stump, G., Yukish, M.A.: Trade Space Exploration of a Wing Design Problem Using Visual Steering and Multi-Dimensional Data Visualization. In: 49th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference. Schaumburg, IL (2008). DOI 10.2514/6.2008-2139
45. Society for Automotive Engineers, Inc.: ARP4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems. Tech. rep., SAE International (1996)
46. Society for Automotive Engineers, Inc.: ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. Tech. rep., SAE International (1996)
47. Stewart, D., Whalen, M.W., Cofer, D., Heimdahl, M.P.: Architectural Modeling and Analysis for Safety Engineering. In: M. Bozzano, Y. Papadopolous (eds.) Proceedings of International Symposium on Model-Based Safety and Assessment (IMBSA 2017), pp. 97–111 (2017). DOI 10.1007/978-3-319-64119-5_7
48. Stump, G., Lego, S., Yukish, M., Simpson, T.W., Donndelinger, J.A.: Visual Steering Commands for Trade Space Exploration: User-Guided Sampling With Example. Journal of Computing and Information Science in Engineering **9**(4), 044,501 (2009). DOI 10.1115/1.3243633
49. Stump, G., Yukish, M., Martin, J., Simpson, T.: The ARL Trade Space Visualizer: An Engineering Decision-Making Tool. In: 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference. Albany, New York (2004). DOI 10.2514/6.2004-4568
50. Stump, G., Yukish, M., Simpson, T., Harris, E.N.: Design Space Visualization and Its Application to a Design by Shopping Paradigm. In: 29th Design Automation Conference, Parts A and B, vol. 2003, pp. 795–804. ASME, Chicago, Illinois, USA (2003). DOI 10.1115/DETC2003/DAC-48785
51. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus. In: Leningrad Seminar on Mathematical Logic, pp. 1–11. Leningrad (1966)
52. Verendel, V.: Quantified security is a weak hypothesis. In: Proceedings of the 2009 workshop on New security paradigms workshop - NSPW '09, p. 37. ACM Press, New York, New York, USA (2009). DOI 10.1145/1719030.1719036. URL `http://portal.acm.org/citation.cfm?doid=1719030.1719036`
53. Watkins, C.: Integrated Modular Avionics: Managing the Allocation of Shared Intersystem Resources. In: 2006 IEEE/AIAA 25TH Digital Avionics Systems Conference, pp. 1–12. IEEE, Portland, OR (2006). DOI 10.1109/DASC.2006.313743
54. Zantema, H., Groote, J.F.: Transforming equality logic to propositional logic. In: FTP'2003, 4th International Workshop on First-Order Theorem Proving (in connection with RDP'03, Federated Conference on Rewriting, Deduction

and Programming), pp. 162—-173 (2003). DOI 10.1016/S1571-0661(04)80661-3
55. Zimmermann, H.J.: Fuzzy Set Theory–and Its Applications, fourth edn. Springer Netherlands, Dordrecht (2001). DOI 10.1007/978-94-010-0646-0