Architecture Needs Behavior

Lee Pike
Galois, Inc.
leepike@galois.com
DRAFT

***Abstract*—I argue that architecture and behavior are inseparable concepts. Traditionally, architecture description languages focus on architecture. Programming languages focus on behavior. Both are necessary to specify and reason about systems. Good old abstraction, that is part of modern programming languages, provides the ability to capture both architectural and behavioral aspects of a system.**

I. Introduction

> *In principle, ADLs [architecture description languages] differ from programming languages because the latter bind all architectural abstractions to specific point solutions whereas ADLs intentionally suppress or vary such binding. In practice, architecture is embodied and recoverable from code, and many languages provide architecture-level views of the system [Cle96].*

Can architecture be specified, reasoned about, and refactored independently of component behavioral implementations? The question hinges on whether architecture can be considered an abstraction of a system, independent of behavior. I argue that it cannot and that trying to decompose them leads to bugs and violated assumptions.

I begin by providing an example in Section II of trying to perform an architectural refactoring in an autopilot my team was building, and I discuss the subtle interactions between architecture and behavior arising in the refactoring. In Section III, I explore potential solutions for trying to ensure that ADLs and behavioral implementations match.

If architecture is not a separate abstraction layer from behavior, what is it? I claim that architecture is a *property* of behavior. Behavior is specified in programs; program properties are naturally encoded as types. I therefore explore the notion of an *architectural type system* for programs in Section IV.

Related work in relating architecture and behavioral specifications is described in Section V, and concluding remarks are made in Section VI.

A. What is an Architecture?

To understand what it means to specify an architecture, we need to know what an architecture is in the first place.

A *component* is an abstract input/output automata. From the view of the architecture, a component is abstract, although it may even have an architecture of its own. *Channels* connect the outputs of one component to the inputs of another; there may be different kinds of channels with different semantics; for example, a channel may have message-passing semantics or shared-state semantics. Components and channels may be dynamically created or destroyed during execution.

Architectural properties are the visible behavior of the components (i.e., their inputs and outputs). These sort of properties are sometimes called *contracts* [Mey92]. Other properties may constrain non-functional properties of the system, including memory and time.

A system's *architecture* is its set of components, its channels, the topology the channels enforce on the components, and the architectural properties of the system.

B. The ADL Vision

The ADL vision is compelling. In designing large systems (or "systems-of-systems"), after requirements are specified, an architecture is designed to meet those requirements. The architecture specifies dataflow between behavioral components, interfaces, and assumed hardware abstractions. The behavioral components are then implemented to satisfy the architectural contracts and meet the specified interfaces. A waterfall [Roy87] style workflow is envisioned, that flows "down" from requirements to architecture to behavior, then flows "up" from behavior to architecture to requirements in verification and validation and integration.

System architects design the architecture. *System architects are not software engineers.* Architects bridge a world requiring interaction with a diverse range of stakeholders ranging across customers, hardware engineers, certification authorities, safety and security engineers, business managers, and finally software engineers.

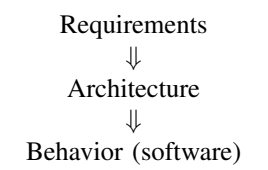

Fig. 1. The architecture abstraction.

Thus, an ADL is envisioned to be a language that bridges the gap between these diverse worlds (Figure 1). It is neither a software specification nor requirements specification. An ADL is a more rigorous, uniform, and analyzable way to capturing system architecture than the typical approach: capturing an architecture via a collection of prose in a word processing document, together with *ad-hoc* figures, or even more likely, never explicitly capturing the architecture in the first place.

Furthermore, at the architectural abstraction level, we do not want to prematurely commit to a particular programming

language, and an architecture may contain components that are implemented using a variety of languages.

Architectures are not static. Like software, they must evolve as requirements and behavioral components evolve. ADLs promise to support *compositional* design and refactoring [MWRH13], [SLNM05]. Once an architectural specification is captured in an ADL, system architects can more easily refactor and respond to changes. An architecture can be analyzed against changing requirements rigorously. New behavioral components can seamlessly be integrated, modified and optimized without considering the full system specification. Indeed, an ADL is agnostic about how behavior is implemented so that an implementation in one programming language may be substituted for another. One needs only check that the component contracts, as specified in the ADL, remain satisfied.

II. The Problem: Architectural Refactoring

I argue that the vision outlined above does not work in practice. In particular, architectural refactoring and component refactoring are intrinsically coupled so that in the general case, refactoring the one requires refactoring the other.

I will argue the point by considering an example inspired by the architecture for a secure autopilot called *SMACCMPilot* [HPE+14]. SMACCMPilot was developed to showcase formal methods technologies to make cyber-physical systems more secure. Consider an implementation of SMACCMPilot's communications and crypto subsystem. We describe two architectural refactorings and show that an architectural refactoring necessitates a behavioral refactoring.

A. Initial Architecture

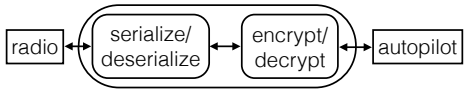

Fig. 2. Initial architecture.

The architecture in Figure 2 depicts a simplified encrypted communications subsystem for an autopilot. The subsystem mediates communications between a ground control station (GCS) and the autopilot itself. Messages to and from the GCS are encrypted. (For simplicity, assume symmetric key cryptography.) A message received on the radio module from the GCS is first deserialized, including error detection and correction. Then it is decrypted and passed to the autopilot for processing. In the other direction, all telemetry from the autopilot is encrypted before being serialized and passed to the radio module for broadcast to the GCS.

The behavior components handle dataflow in two directions: from the autopilot to the GCS and from the GCS to the autopilot. Suppose then that each component is implemented as a multi-threaded program. For example, in the `block cipher` component, one thread listens for messages coming from the `serialize/deserialize` component, calls a `decrypt()` function with incoming messages, and places the results on an interface to be sent to the autopilot. The other thread listens to the autopilot databus, calls an `encrypt()` function with incoming messages, and then places the result on the outgoing interface for broadcast to the GCS. The `serialize/deserialize` component has a similar multi-threaded implementation.

B. A First Refactoring

After building and testing the system, some infelicities in the design are found:

- The multi-threaded software components are notoriously difficult to debug and may suffer deadlocks and livelocks.
- Also, supposing each component is a process executing on an RTOS, by decomposing the components, all concurrency is "flattened" to be at the top-level, simplifying scheduling and concurrency reasoning.
- Encryption and decryption are part of the same behavioral component. The system can be made more secure by decomposing the crypto component into separate encryption and decryption components. That way, if the encryption (or decryption) function is flawed or usurped so that communications from the autopilot to the GCS (from the GCS) are prevented, it does not affect communications from the GCS to the autopilot (or vise versa).
- With encryption and decryption decomposed, we can lift an implicit assumption that encrypt and decrypt use the same symmetric keys; different keys can be used in different directions [DuB13]—one key for communications from the GCS to autopilot and a separate key from the autopilot to the GCS.
- Moreover, once encryption and decryption are decomposed, one can see that we can keep incoming and outgoing messages completely separated by decomposing the `serialize/deserialize` component, too.

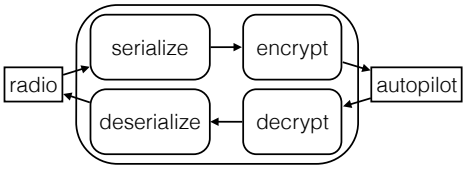

Fig. 3. Refactored secure architecture.

Based on these changes, a refactored architecture is shown in Figure 3.

The new architecture needs fundamentally new software components. Creating the revised software is not simply a matter of decomposing some functions and declarations in one module into two. Rather, multi-threaded programs have to be made single threaded. Marshaling code that places output onto an interface has to be refactored. Additionally, we need to ensure that no legacy behavior or libraries are left behind a decomposed module. Finally, depending on the implementation, it may not be easy to ensure that a composed component that internally communicated via shared state now no longer does so. For example, if the implementation does not guarantee memory isolation between processes, then shared-state communication can occur between processes.

C. A Second Refactoring

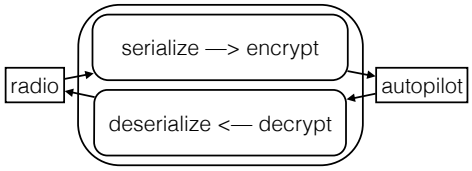

Fig. 4. Efficient refactored architecture.

In system integration testing, suppose we find that the Figure 3 architecture is no longer schedulable given that we have increased the number of processes from two to four. Therefore, an alternative design is proposed that composes components orthogonally to how we have done so in Figure 2. This time, we compose the deserialization and decryption components, and compose the serialization and encryption components, as shown in Figure 4. The new architecture and components have the following desirable characteristics:

- There are two processes to schedule.
- The composed behavioral components do not need to be concurrent, since dataflow is unidirectional through each component. For example, after decryption, serialization can be called directly in a single-threaded program.
- The property of separating incoming and outgoing communications is maintained.

This third architecture also requires refactoring the software once again, this time, merging systems. Communication through the architectural interfaces now become shared memory and function calls. Care must be taken to ensure that the new software does not continue to publish messages to an unused and undocumented interface that could maliciously be used by other components.

III. A Tale of Two Languages

When an ADL is a separate language than the programming languages used to implement the behavioral components, we lose a formal connection between them. The architectural specification can drift away from the actual implementation. Put another way, solving the architectural modeling problem by introducing yet another language means you now have two problems: ensuring that the program implementing the architecture is correct (the original problem), and ensuring that the ADL accurately represents that implementation.

The two language problem is not unique to ADLs; the criticism has also been leveled against formal methods approaches in which formal models do not match the implementation [Pto14, p. 10][AP11], [Smi85] or are even inconsistent [Pik06].

There are at two approaches for connecting ADL specifications to behavioral implementations: component contracts and glue code generation. We consider each in turn.

A. Component Contracts

Component contracts are compositional: so long as the rest of the system satisfies the assumptions on which the component relies, we can verify or validate that the component guarantees the contracts on its behavior. A component can thus be tested or even proved correct in isolation from the remainder of the system, or at least in the context of an abstracted or simulated system. Specification languages for stating behavioral contracts have been developed; examples include AGREE [CGM+12] and BLESS [LCH13] in AADL.

But how can we ensure that a component actually satisfies the constraints? That requires verifying the component implementation, which requires breaking the abstraction boundary between architecture and behavior. In a system refactoring in which the component behavior and architecture change in tandem, the contract modification and component refactoring naturally occur in tandem.

Non-functional properties present an additional problem. Prototypical non-functional properties are timing and memory usage. These sort of properties are often considered architecture-level properties, because they affect the composition of behavioral components. For example, the schedulability of multiple components on a single processor is commonly considered an architecture-level property. But schedulability depends on each component satisfying timing constraints. For example, at the architectural level, a component might be given a worst-case execution time (WCET) of 100 milliseconds to ensure it is schedulable.

But WCET itself of the component is only meaningful in the context of the full system. It depends on the hardware (e.g., processor speed, caches), the other components (e.g., their processor locks and priorities), the operating system (eg., scheduler and context switching time). There is no simple way to abstract out the rest of the system to validate non-functional properties.

B. Glue Code

Another approach to connecting the ADL with behavioral implementations is via so-called "glue code" generated from an ADL [FGA+13]. Glue code can include software for implementing the communication ports between behavioral components and binding behavioral components to a platform (e.g., process creation code on an operating system). Not only does glue code generation relieve the programmer from the tedious task of writing "boilerplate" code, it provides some evidence that the behavioral components are consistent with the architecture.

But this consistency fundamentally depends on how well an *implementation* enforces consistency between program fragments. For example, one ADL I know generates C code for the interfaces. Because of build system constraints, the interface code and the behavioral components (also in C) are compiled separately then linked to create an executable.

But typical linkers do not enforce consistency across object files. Suppose that `interface.c` from Figure 5 implements an interface generated by an ADL, and `main.c` implements a consumer of the interface. Using `gcc -Wall` to compile each file separately then linking them together produces no warnings and an executable program that has undefined behavior (the result returned on my computer after a particular compilation is `1343657516`).

```
int foo(int a, int b) {
  return a+b;
}
```

```
int foo(int a);

int main(void) {
  printf("%d\n", foo(42));
  return 0;
}
```

Fig. 5. Interface (`interface.c`, top) and client (`main.c`, bottom) code mismatch.

While other languages may provide stronger guarantees when composing program fragments, the resolution is at the abstraction level of the software implementation, not the ADL.

Furthermore, glue code can violate non-functional assumptions such as stack usage or timing, and the developer has no insight into the glue code impact on system design. For example, consider an example that arose in SMACCMPilot, with a component as shown in Figure 6:

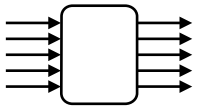

Fig. 6. Fan-in, fan-out.

A component had a large number input connections ("fan-in") as well as output connections ("fan-out"). The input ports were incoming serialized messages, tagged with an identifier. The output ports were for unpacked messages, one port for each kind of message. We generated glue code from AADL [FG12] specifications. In the glue code implementation, for each input port, memory is statically reserved for each output port the incoming data may be eventually sent out on (the reason for doing so has to do with the concurrency semantics implemented, and are not important for this discussion). So if every input port can potentially send to each potential output port, then $i \times o$ memory slots are reserved, where i are the input ports and o are the output ports.

The component had dozens of input ports and output ports. A crash was encountered during integrated testing that we could not attribute to application code. Eventually, it was traced to the glue code: a process implementing the component allocates memory on its stack for input and output ports. The cross-product of ports became so large that the process blew its stack!

But our design intention was that each input port correspond to exactly one output port. By providing the additional information to the architectural model, the $i \times o$ memory slots were reduced to i (or o, since there are exactly as many input as output ports) slots.

Glue code generation, while useful, cannot provide a sufficient connection between architecture and behavior.

IV. Architecture as Types

So far, I have argued the perils of decomposing architectural from behavioral specifications. The reason is that architecture is not a separate abstraction layer from behavior. Rather, I argue that architecture is a *property* of programs. An established method for tying properties to programs is via *types*, and so the way to approach architectural specifications is via an *architectural type system.*

I first give an example of how an architectural and component programming language can be combined, using types to tie them together. Then I describe type system research on which an architectural type system could be based. I then describe remaining challenges.

A. Composing Architecture and Behavior

An approach to composing architectural and behavioral programs was taken in implementing portions of SMACCMPilot, mentioned in Section II. Behavioral components are built using Ivory [EPW+15], a "safe-C" language, designed for low-level embedded programming but that guarantees the absence of the vulnerabilities found in C. The architecture is built using Tower [HPE+14] that provides architectural constructs; namely, the notion of components, hardware interfaces, and communication channels between components.

Ivory and Tower are both embedded domain-specific languages (EDSLs) [Gil14], meaning they are embedded within a host language; in this case, the host language is Haskell [Has]. EDSLs have the advantage that their host-language acts as a Turing-complete meta-programming language for the embedded language. By having the same host language, Ivory and Tower programs share the same type system and meta-programming language so they are not so much different languages as aspects of a single language.

Tower types describe data passed over channels and ensure the types of state-machines (implemented in Ivory) match the inputs and outputs of channel data. Tower channels are represented as input/output pairs. The first element of the pair is the input portion of the channel, allowing a producer to place data into the channel, and the second element is the output portion, allowing a consumer to pull data off a channel.

```
(tx, rx) :: (ChanInput a, ChanOutput a)
```

For example, the pair `(tx, rx)` is a pair that has the type `(ChanInput a, ChanOutput a)`, where `tx` is a handle to place values in a channel and `rx` is a handle to pull values from a channel.

A *handler* in Tower takes a receive end of a channel, and an Ivory program to run on the received value. The Ivory program is a *callback*, or anonymous function to run. So Ivory/Tower programs have the form

```
handler rx ...
  callback ( \val -> ... )
```

where `callback` takes a lambda term as an argument. If `rx` has type `ChanOutput Integer`, then `val` has type `Integer`.

In Tower, channel endpoints are values that can be created and manipulated. *Executing* a Tower program *generates* an architecture. Arbitrary type-safe, Turing-complete computation can be done when generating the architecture, including, for instance, specializing the number of components depending

on compile-time constraints, and adding debugging channels for testing. We have done both.

For example, we have a single-board autopilot implementation that executes solely on an ARM Cortex M4-based board, which is used for testing and development, and a dual-board design that also uses an ARM Cortex A15-based board which communicates with the ARM Cortex M4 board over a CAN bus. Depending on compile-time options, either the first or second architecture is compiled from the same code. We do not have to maintain two models or perform a manual refactoring to get from one to the other.

Once the architecture is generated, it is fixed. The architecture specification generated from a Tower program is essentially isomorphic to an AADL specification; indeed, AADL specifications can be generated from Tower [HPE+14].

B. Toward an Architecture Type System

ADLs can provide a formal model of a system, and in a formal system, formal properties can be proved. Toward that end, a logic is associated with the ADL for specifying and proving properties [CGM+12], [LCH13].

The Curry-Howard isomorphism provides a correspondence between logics and type systems [Wad15]; the slogan of the correspondence is that "propositions are types" and "proofs are programs". The isomorophism is a powerful idea because it means that any logic has a corresponding type system, and vice versa.

Specialized type systems can describe protocols and message-passing systems. One such type system is *session types*, a type-system extension developed to statically prove properties about protocol implementations [HVK98], [DCD10], [GVR03]. The basic idea is that each state of the protocol is modeled as part of the type of the channel it is performed on, with that type being mutated by actions over it. Session types can also describe branching and choice, allowing for failure at different points in a protocol, or for a client to make choices about what the next state should be.

In contrast to many ADLs, they can specify not only channels and their types, but the message-passing behavior (or protocols) over the channels, doing so statically.

A standard example to explain session types is one in which a client provides a server two numbers, the server adds them and returns them to the client, closing the connection [GVR03].

First, let us create a session type for this action, from the perspective of the server. Let $?a$ represent an input argument of type a, $!b$ represent an output argument of type b, $a.b$ represent a sequence of two operations, and End the closing of a connection. Then the following session type corresponds to the program described above:

$$?Integer.?Integer.!Integer.End$$

The type does not describe the computation performed by the server; it only describes the expected sequence of actions for the protocol. If an implementation subtracted the first number from the second or returned the first number and ignored the second, it would still have the above type.

A client has the corresponding dual type:

$$!Integer.!Integer.?Integer.End$$

A program that is typed with session types connects the architecture to the behavior through type-checking. If the implementation does not satisfy the types, type-checking fails. Furthermore, the types can sometimes be inferred from the program.

```
server = offer $ ixdo
  x <- recv
  y <- recv
  send (x + y)
  close
```

```
client x y = ixdo
  send x
  send y
  z <- recv
  close
  ret z
```

Fig. 7. Message-passing program with session types.

For example, consider the implementation in Figure 7, written in the `simple-sessions` language, a domain-specific language for session types in Haskell [PT08]: (For readers familiar with Haskell, `ixdo` depends on a preprocessor to use "do notation".) The `server` program implements the server described above that adds two numbers and returns them. The `client` takes two integers `x` and `y` as arguments and sends them to the server, awaiting a response.

The types of `server` and `client` are respectively computed automatically by type inference. Let me repeat: one needs only to write the programs above, and the type system computes the types. Taking our position that types denote an architecture, the type system infers an architecture from a program!

C. Architectural Type System Challenges

While I advocate for "architecture as types", the approach is not without challenges, and session types represent a single point in the design space.

Large systems often contain components implemented in multiple programming languages, but a type system is associated with a single language. Indeed, one of the motivations of an ADL is as a unifying language, since an architecture may deal with components from multiple languages. So how does a type system help us?

Programmers deal with multiple programming languages through interfaces. A foreign function interface allows one to directly access functions from one language in another. Typically one language is primary in the software architecture—but that does not obviate the use of other languages. An ADL is a language itself, so the idea that an ADL is language-agnostic is specious; an ADL may be agnostic with respect to programming languages—since an ADL is not one—but it is not language agnostic.

Another challenge is being able to reason about non-functional properties such as timing and memory usage, and

being able to do so through different levels of abstraction. While I present no particular insights into how to solve this problem (but see Edward Lee's article [Lee09]), I have argued in this paper that these properties cannot be addressed outside of an implementation.

Architecture is an abstraction, but an architecture itself can have levels of abstraction, e.g., refining a fault-intolerant architecture into one that masks faults [BK07]. One form of abstraction is something I will call *architectural polymorphism*. An example of architectural polymorphism is to keep the number of nodes in a distributed system uninterpreted as n nodes, where n can be instantiated with specific values. Distributed algorithms are typically designed for arbitrary numbers of nodes (with constraints). However, ADLs typically have few polymorphism capabilities. For example, the popular ADL AADL [FG12] does not support node polymorphism.

High-level programming languages, however, have polymorphism built in as a cornerstone of program abstraction. Still, specific support for architectural polymorphism over nodes, channels, protocols, replication, etc. is needed.

A type system typically has constrained expressiveness to preserve (mostly) decidable type inference. Dependent types are more general, but type inference is lost [Wad15]. In our example of session types, some properties are not expressible; for example, one cannot assert that an integer returned by a server is greater than the two integers sent to it.

Refinement types are more expressive types that maintain decidability via techniques like enforcing structural type recursion [FP91] or *logically qualified data types*, abbreviated as *liquid types*, a type system that combines Hindley-Milner type inference with decidable logical theories (*satisfiability modulo theories*) [RKJ08]. Refinement types for ADLs could be a promising avenue of research.

V. Related Work

Our claim is related in spirit—and this paper's title pays homage to—the position paper entitled, "Computing Needs Time", arguing that time, a non-functional property, should be a first-class aspect of programs [Lee09].

The "Twin Peaks" model argues that requirements and architecture be developed and refined in tandem [Nus01]. The Twin Peaks model is motivated by the claim made by Swartout and Balzer that specification and implementation fundamentally cannot be independent or sequential [SB82]. The problem, they argue, is that missed assumptions and constraints nearly always require changes to a specification during implementation. Today, one sees the upshot of these arguments in practice: the Waterfall model of software development is largely abandoned. The Waterfall development proposes that in requirements, architecture, and implementation be developed in series, and each be finalized and frozen before moving onto the next.

Two modern ADLs that connect architecture specifications to implementations are the *P* language [DGJ+12] allows one to specify both architecture and components in a single unified language. It includes a model-checker backend. *P* has an asynchronous semantics. Another domain-specific language for specifying distributed algorithms that can be compiled to executing code is *DistAlgo* [LLS11].

For a general survey of ADLs, I refer the reader to Tomiyama *et al.* [THG+99] and Medvidovic and Taylor [MT00].

VI. Conclusion

The fundamental thesis put forth here are that (1) ADLs are incomplete without being tied to behavioral implementations, and (2) architecture is best considered as a property of behavior (and types naturally capture program properties).

Wither architecture description languages? Hardly. ADLs provide many practical benefits not addressed by programming languages. For example, significant modeling and analysis tools have been built over ADLs. Taking AADL as a particular example, there is a real-time scheduling analysis [SLNM05], assurance-case analysis [GBC+14], and model-checkers [CGM+12], for example. AADL provides a common language on which to build a range of analyses focused an the architectural level.

Additionally, in practice, ADLs are not competing with type theories, they are competing with Word documents, the dominate means by which architecture, interfaces, and constraints are modeled. Informal, ad-hoc documents suffer the criticisms made herein far worse than ADLs do.

Acknowledgements

I believe many of my colleagues, funders, and collaborators would actively disavow themselves of these viewpoints. Good ideas have been borrowed and cited; bad ideas are mine alone.

References

[AP11] Darren Abramson and Lee Pike. When formal systems kill: computer ethics and formal methods. In *APA Newsletter on Philosophy and Computers*, volume 11. American Philosophy Association, 2011.

[BK07] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *International Conference on Distributed Computing Systems (ICDCS*, page 3, 2007.

[CGM+12] D. Cofer, A. Gacek, S. Miller, M. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In *Proceedings of NASA Formal Methods*, pages 126–140, 2012.

[Cle96] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE Computer Society, 1996.

[DCD10] Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. Sessions and session types: an overview. In *Proceedings of the 6th international conference on Web services and formal methods*, pages 1–28. Springer-Verlag, 2010.

[DGJ+12] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. Technical Report MSR-TR-2012-116, November 2012.

[DuB13] Thomas M. DuBuisson. SMACCMPilot secure MAVLink communications. Technical report, Galois, Inc., 2013. Available at http://smaccmpilot.org/artifacts/Galois-commsec.pdf.

[EPW+15] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. Guilt free Ivory. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 189–200. ACM, 2015.

[FG12] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[FGA+13] Matthew Fernandez, Peter Gammie, June Andronick, Gerwin Klein, and Ihor Kuz. CAmkES glue code semantics. Technical report, NICTA and UNSW, Australia, nov 2013.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Progamming Language Design and Implementation (PLDI)*, 1991.

[GBC+14] Andrew Gacek, John Backes, Darren D. Cofer, Konrad Slind, and Mike Whalen. Resolute: an assurance case language for architecture models. In *Proceedings of High integrity language technology, HILT*, pages 19–28, 2014.

[Gil14] Andy Gill. Domain-specific languages and code synthesis using Haskell. *Queue*, 12(4):30–43, April 2014.

[GVR03] Simon Gay, Vasco Vasconcelos, and Antonio Ravara. Session types for inter-process communication. Technical Report TR-2003-133, University of Glasgow, March 2003.

[Has] Website. https://www.haskell.org/.

[HPE+14] Patrick C. Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. Building embedded systems with embedded DSLs (experience report). In *Intl. Conference on Functional Programming (ICFP)*. ACM, 2014.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, pages 122–138, London, UK, 1998. Springer-Verlag.

[LCH13] BrianR. Larson, Patrice Chalin, and John Hatcliff. BLESS: Formal specification and verification of behaviors for embedded systems with software. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 276–290. Springer, 2013.

[Lee09] Edward A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, May 2009.

[LLS11] Yanhong A. Liu, Bo Lin, and Scott D. Stoller. Programming and optimizing distributed algorithms: An overview. In *Proc. 8th International Conference & Expo on Emerging Technologies for a Smarter World (CEWIT 2011)*. IEEE Press, November 2011.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), 2000.

[MWRH13] Anitha Murugesan, Michael W. Whalen, Sanjai Rayadurgam, and Mats P.E. Heimdahl. Compositional verification of a medical device system. In *ACM International Conference on High Integrity Language Technology (HILT) 2013*. ACM, November 2013.

[Nus01] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–117, March 2001.

[Pik06] Lee Pike. A note on inconsistent axioms in Rushby's "Systematic formal verification for fault-tolerant time-triggered algorithms". *IEEE Transactions on Software Engineering*, 32(5):347–348, May 2006.

[PT08] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *ACM SIGPLAN 2008 Haskell Symposium*, Sept. 2008.

[Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169. ACM, 2008.

[Roy87] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, Intl. Conference on Software Engineering, pages 328–338. IEEE, 1987.

[SB82] William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

[SLNM05] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with AADL. In *Proceedings of the 2005 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time &Amp; Distributed Systems Using Ada and Related Technologies*, SigAda '05, pages 1–10. ACM, 2005.

[Smi85] Brian Cantwell Smith. The limits of correctness. *SIGCAS Computer Society*, pages 18–26, January 1985.

[THG+99] Hiroyuki Tomiyama, Ashok Halambi, Peter Grun, Nikil Dutt, and Alex Nicolau. Architecture description languages for systems-on-chip design. In *Sixth Asia Pacific Conference on Chip Design Language*, pages 109–116, 1999.

[Wad15] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, November 2015.