Unmanned Autonomous Verification and Validation

Position Paper

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

Don Stewart
Galois, Inc.
galois.com
dons@galois.com

John Van Enk
DornerWorks, Inc.
dornerworks.com
John.VanEnk@dornerworks.com

ABSTRACT

We outline a new approach to the verification and validation (V&V) of safety-critical avionics based on the use of executable *lightweight domain specific languages* (LwDSLs)—domain-specific languages hosted directly in an existing high-level programming language. We provide examples of LwDSLs used in industry today, and then we describe the advantages of LwDSLs in V&V. We argue the approach promises substantial automation and cost-reduction in V&V.

1. INTRODUCTION

Next-generation unmanned air vehicles (UAVs) will contain highly-complex software, as human ability and judgment is replaced by software systems. In addition, UAVs will be expected to coordinate with piloted aircraft, ground systems, and even other UAVs. This new functionality requires the specification and implementation of complex new software systems in new design domains—for inter-UAV coordination, ground-system coordination, UAV autopilot, pilot artificial intelligence systems, internal health-management and more. As a result, not only will the size and complexity of individual software systems increase but so will the complexity of the interactions between software systems in different design domains.

Verification and validation (V&V) approaches to manage this engineering effort *must* keep pace with both challenges. There is a need then, we argue, for "unmanned and autonomous" approachs to V&V— techniques that will make tractable the exponential growth in complexity of UAV systems by taking advantage of new research in the automation and mechanization of V&V.

In system design, a proven technique for managing complexity, and gaining abstraction is through *domain-specific languages* (DSLs)—languages tailor-made to describe the concepts of a particular design space. A DSL exposes the abstractions of the domain to the programmer, relieving them from having to consider irrelevant detail. For example, a simple and well-known example of a DSL is the `Yacc` parser generator (for writing the front-end of compilers). The `Yacc` language is a stylized Backus Normal Form in which programming language syntax is naturally expressed. `Yacc` then compiles the BNF specification to `C` code.

Programming in a good DSL is more like writing an executable specification than writing a program. The DSL relieves the developer of boilerplate programming issues. Users of Yacc, for example, write directly in the BNF specification notation, removing the need to translate the grammar to a hand written parser in some concrete implementation language. High-level DSLs, in effect, serve as the *executable requirements* for an implementation. This DSL specification can then in turn be used for modeling, simulation, and synthesis.

Because of the variety of problem domains next-generation UAV software must address, no single DSL can cover all requirements. Instead we suggest a *family of DSLs*, each appropriate to its domain. However, if designing a DSL means building a new language, compiler, and V&V tools specifically for the DSL from scratch, the "DSL approach" would be cost-prohibitive given the multitude of problem domains that must be addressed.

There is a better way: *lightweight domain-specific languages* (LwDSLs) have been quietly gaining traction in industry.[1] A LwDSL is a DSL hosted in a high-level general-purpose language, allowing us to reuse all of the infrastructure provided by a mature language to implement a specific DSL.

Many LwDSLs—and all of the ones we describe in this paper—are hosted in the popular functional programming language, Haskell. A high-level functional language such as Haskell makes it easier to construct domain-specific functions, libraries, and even syntax, as well as being more amenable to verification processes. So by using a LwDSL, a domain expert enjoys the benefits of the DSL approach in having the right level of abstraction, while gaining access to the host language's existing compiler, libraries, and validation tools... almost *for free*.

2. LIGHTWEIGHT DSLS IN PRACTICE

LwDSLs have been successfully used in industry for hardware and embedded software design. The following are some examples:

Jones employed a LwDSL for configuring large-scale, real-time embedded systems for Boeing, showing significant improvements over previous approaches with reduced code size,

[1] LwDSLs are also referred to as *embedded* DSLs (EDSLs or DSELs) in the literature [5].

CPS Week Workshop on Mixed Criticality 2009, San Francisco, California, USA

increased modularity and scalability, and easier, earlier detection of defects [6].

Hawkins described a LwDSL used at Eaton to intuitively describe the safety-critical behavior of embedded code for hydraulic hybrid vehicle control, lowering the risk of introducing bugs in the design phase. They describe this approach as "RTOS Synthesis", automating most of the work of a real-time operating system, with increased assurance [4].

Antiope employed a similar strategy for the design of ultra-low power radio chips. Their LwDSL played two roles: it was the main language for simulation used to debug their protocol designs, and it was also the implementation language for their verification tools [7].

In partnership with Chalmers University, Xilinx designed a hardware description language that provides high-level abstractions and aids in proving circuit equivalence [1].

These are just some examples of industrial LwDSLs; companies in a variety of industries are continuously realizing their cost-effectiveness and assurance guarantees.

3. UNMANNED AUTONOMOUS V&V

As we have mentioned, V&V must scale with software complexity, to become what we call "unmanned autonomous" V&V (UAV&V). In this section, we describe some of the tools a good host language makes available to LwDSLs (we use Haskell as our running example host) that make UAV&V possible, including invariant enforcement, automatic test-case generation, and coverage analysis.In addition, LwDSLs are used not only for V&V activities like modeling, testing, and simulation but also for direct synthesis of executable code, so we mention tools enabling synthesis.

V&V Tools.

Semantic Types: By using Haskell to host domain-concepts, we can reuse the significant effort required to construct a sophisticated *static* type system—which is one of the cheaper ways to gain static assurance against a variety of defects. The Haskell type-system is particularly powerful and expressive and can be used to ensure deep program invariants hold at compile-time.

Automated Testing & Coverage Analysis: Tools are available that automate testing and coverage analysis of the host language. QuickCheck [2], for example, allows one to embed properties in Haskell programs and automatically generate random data (that meets coverage criteria) to test those properties. One writes properties about Haskell programs (or hosted LwDSLs) in Haskell directly, which can in turn face coverage analysis via tools such as HPC [3].

Libraries and Support: The popular host language lives in a much larger ecosystem than a domain-specific language. Haskell has over 1,000 released open-source libraries at the time of writing—effort that could not be duplicated using only a DSL approach. In addition, the host language's foreign function interface eliminates the need to rewrite existing libraries and code, cutting risk and costs.

Formal Verification Tools: Mechanical theorem-provers, model checkers, and automated solvers (e.g., decision procedures) are essential V&V tools for ultra-critical systems. Haskell has libraries and tools that make it easy to translate a hosted LwDSL into those tools. Furthermore, as a functional language itself, Haskell is naturally amenable to mathematical modeling and analysis.

Synthesis Tools.

Code Synthesis Tools: Along with tools for testing and coverage, a lightweight DSL approach saves effort through the transparent reuse of techniques and tools for code generation and synthesis from the host language (such as C generation libraries and tools), making synthesis cheaper.

Portability and Maintainance: An LwDSL also allows us to gain improve maintainance and portability, as the LwDSL needn't commit to any particular architecture, instead being as cross-platform as the host language.

4. CONCLUSIONS

LwDSLs will not make the challenges of mixed-criticality systems go away. In particular, the systems must be designed from the outset to be modular and compositional. The right architectural abstractions must be made from the outset to ensure time and space partitioning, fault-tolerance, and security. However, LwDSLs have already proven themselves in other related industries. We believe they are an essential part of the solution to cost-effective V&V for next-generation UAV systems.

About the Authors

Dr. Lee Pike is a Senior R&D Engineer at Galois, Inc. specializing in formal methods for critical systems. Previously, Dr. Pike was a staff scientist at the NASA Langley Research Center. He has a best-paper award from *Formal Methods in Computer-Aided Design* in 2007.

Don Stewart is a Senior R&D Engineer at Galois, Inc. specializing in functional languages. He is the coauthor of the popular textbook *Real World Haskell* and has a best-paper award from *Practical Aspects of Declarative Languages*, 2007.

John Van Enk is a software engineer at DornerWorks specializing in safety-critical applications, embedded software, and certification.

5. REFERENCES

[1] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.

[2] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of the ICFP*. ACM SIGPLAN, 2000.

[3] A. Gill and C. Runciman. Haskell program coverage. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 1–12. ACM, 2007.

[4] T. Hawkins. Controlling hybrid vehicles with Haskell. In *Proc. ACM CUFP '08*, New York, NY, USA, 2008. ACM.

[5] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[6] M. P. Jones. Experience report: playing the DSL card. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 87–90, New York, NY, USA, 2008. ACM.

[7] G. Wright. Functions to junctions: ultra low power chip design with some help from Haskell. In *Proc. ACM CUFP '08*, New York, NY, USA, 2008. ACM.