Practical Formal Verification of Domain-Specific Language Applications

Greg Eakman[1], Howard Reubenstein[1], Tom Hawkins[1], Mitesh Jain[2], and Panagiotis Manolios[2]

[1] BAE Systems, Burlington MA 01803, USA
[2] Northeastern University, Boston MA 02115, USA

Abstract. An application developer's primary task is to produce performant systems that meet their specifications. Formal methods techniques allow engineers to create models and implementations that have a high assurance of satisfying a specification. In this experience report, we take a model-based approach to software development that adds the assurance of formal methods to software construction while automating over 90% of the formal modeling. We discuss a software development methodology and two specific examples that illustrate how to integrate formal methods and their benefits into a traditional (testing-based) software development process.

1 Introduction

Domain-Specific Languages (DSLs) provide expressive semantics for defining computational behavior while insulating the application developer from many sources of error inherent in programming in the underlying target programming language (e.g., C or Java). A DSL can be transformed using a code generator into a traditionally compilable source language where the full expressivity of that language can be used, assuming that the transformation technology respects the execution semantics of the target language (Figure 1). The semantics of the DSL, despite the intentions of the DSL designer, are really defined in the translation rules of the code generator. In this paper, we apply formal semantics to two DSLs: a Haskell-based text DSL (Ivory developed by Galois) and a graphical DSL based on a UML profile. We augment the code generation strategy to include generation of a shallow embedding of the target DSL program in ACL2s. We prove application level properties of the DSL program and demonstrate correspondence of the ACL2s model with the source code implementation through testing for equivalent behavior.

This work is motivated by the difficulties observed in accelerating the adoption of formal methods techniques in the practicing software engineering community responsible for deploying systems. The barriers to application developers using formal tools to get their jobs done include: the need to build a constructive proof of correctness before extracting an implementation, unfamiliarity with the syntax and development methodology of formal languages, the difference in semantic models (imperative versus logical/inductive), the difficulty in extracting implementations that interface with existing systems (e.g., those written in C, C++, Java), and the difficulty in obtaining performant implementations.

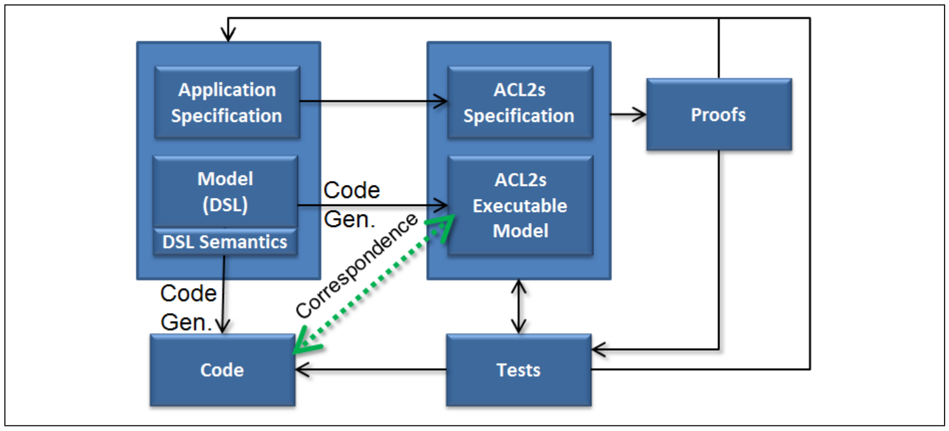

Fig. 1: Properties of the application, developed in the model and proven in ACL2s, are shown to hold in deployed code through correspondence testing.

2 FORMED and UML

Project FORMED (Formal Methods Engineering Desktop)[3] combines software design in the Unified Modeling Language (UML) with formal methods in a way that allows application developers to create formal models. The FORMED DSL is built on the fUML profile [7] which provides executable semantics to class, operation, and association UML elements, and includes semantics for a model-level programming language [2]. This executable UML profile is a graphical DSL, albeit for a very broad domain. From this profile, we build on existing model transformation and code generation tools such as PathMATE [11] to generate a shallow mapping of an application model into ACL2s, the ACL2 Sedan [5]. ACL2s extends ACL2, an industrial strength, semi-automated theorem prover, with a powerful termination analysis engine, a data definition framework, and counterexample generation capabilities. Properties of the application are then specified and proven using ACL2s. We focus on application correctness as part of high assurance software development, rather than on general properties of memory safety or security, which depend on the implementation language, compiler, operating system, and CPU architecture.

The execution semantics of the FORMED UML profile operates on a stack and a heap, modeled in ACL2s with a reusable *context* data definition. The context and the functions that operate on it form the platform model that supports the operation of the UML semantics and the execution of the application. Each context function has an associated input-output contract (proved automatically by ACL2s) that provides assurance on the correctness of the function implementation. We specify and prove many other theorems that axiomatize the semantics of operations on the context. These theorems assist us in abstracting the reasoning about UML semantics. This hierarchical approach to reasoning not only provides a reasoning structure but it also reduces the time taken by ACL2s. For

[3] Sponsored by OSD under contract FA8750-14-C-0024

example, it reduced a proof of an application level property from 20 minutes to less than 15 seconds.

The code generator produces ACL2s code that stores the application state in the stack/heap context. The imperative nature of the FORMED profile maps to ACL2s's functional language by providing the context as one of the input arguments for an operation and requiring the operation to return an updated context. Templates map the UML elements to ACL2s constructs. A UML class, for example, gets mapped to an ACL2s data definition using the defdata framework, which supports automated type-like reasoning [6]. Rather than validate the transformation, we generate theorems based on the UML profile's semantics, such as inheritance, associations, pointers, and class extents, to reason about the ACL2s code.

We have identified common properties of application models and formalized the semantics of each into a theorem template to produce an instance of the theorem for each model location where the property holds. For example, all rooms within a hotel must have a unique room number. The application developer marks the number attribute of the Room class with an «Identifier» stereotype indicating that all instances should have a unique value. We generate an ACL2s theorem that proves by induction that this invariant holds for all operations from any valid state of the application. The code generator statically analyzes the application to assist theorem proving. The unique room number invariant can only be invalidated by operations that write the room number attribute or create a new room. Thus, our proof only needs to consider those critical functions identified through static analysis.

Hotel Locking Example Metrics		
Source UML	UML Classes	13
	Lines Action Language	220
	Operations	26
Java	Generated Java	3895
ACL2s Context	ACL2s SLOC	1284
	Theorems	132
	SLOC Theorems	998
Generated UML Semantics	Executable Code	5652
	Theorem Code	4078
	Theorems Generated	175
Hand-written ACL2s	Application Theorems	18
	SLOC Theorems	806

Table 1. Just 6% of the Hotel application's formal verification code is hand-written.

FORMED proves properties about the shallow embedding of the application in ACL2s, but the formal model is not the deployed code. DSLs, including this UML profile, can generate source in multiple languages (Java in this case), that are deployed as a real application. We build an assurance case argument based on correspondence testing that the properties proven in the ACL2s representation also exist in the deployed code. Since both the ACL2s and deployed code are executable representations generated from the same model, and since we apply the same test cases to each version and verify the corresponding results, we gain confidence that the properties also hold in the deployed code.

We have applied this process to a model of the Hotel Locking problem [9], which describes a protocol for managing room keys to secure hotel rooms between guests. Some metrics from the Hotel application are shown in Table 1. Preliminary results indicate that only about 6% of the ACL2s code, the Application Theorems, need to be hand written. Specifying these theorems in ACL2s requires only moderate knowledge of formal logic. Most of the theorems were automatically proved by ACL2s and some of them required us to guide the theorem prover with appropriate *lemmas*.

3 SITAPS and Ivory DSL

Ivory is a DSL undergoing development by Galois for UAV control systems used on DARPA's HACMS program. Ivory code is currently flying on a quadcopter UAV, running flight control algorithms and data link processing.

Embedded in Haskell, Ivory relies on Haskell's type system to ensure type and memory safety of the generated runtime code (Ivory compiles to C). Though it is a language similar to C, Ivory purposely limits expressiveness to provide memory safety. For example, Ivory does not provide pointer types, nor does it allow arbitrary pointer operations, but it does provide mutable references that are stack allocated and can be passed as procedure arguments. References are ensured not to escape the enclosing stack frame, which is enforced by Haskell types. To prevent buffer overflows, Ivory provides indexing types and bounded loop operations to ensure array accesses are within bounds.

To increase assurance of Ivory programs, we developed a proof framework[4], based on a compilation of Ivory into ACL2s, that verifies user specified and compiler generated assertions. In addition to assertions, Ivory also provides input and output contracts on procedures. Using these contracts to abstract procedure calls, this proof framework is able to perform an efficient interprocedural analysis that scales well with larger programs. The analysis walks through each procedure, generating verification conditions (VCs) for each assertion, each sub input contract at every sub procedure call, and each output contract at every return point. These VCs, which are captured in a VC DSL, are optimized and translated to ACL2s for verification. Verified assertions are removed from the program to lower the runtime overhead and those that fail to prove remain in place to serve as runtime checks. An example Ivory procedure and translation are illustrated below:

```
retractLandingGear :: Def ('[IBool, Sint32] :-> ())
retractLandingGear = proc "retractLandingGear" $
 \ weightOnWheels airspeed -> body $ do
    ifte_ (iNot weightOnWheels .&& airspeed >? 120) -- Better than spec
      (do
        assert $ iNot weightOnWheels  -- Spec guarding
        assert $ airspeed >? 80       -- call to GearUp
        call_ commandLandingGearUp
        retVoid)
    retVoid
```

[4] Supported by DARPA under the SITAPS project under contract FA8750-13-C-0240

Procedure Translated to VC DSL to Verify the First Assertion:

```
let stack0=[] in                          -- Initial stack.
forall free0 in                           -- Free variables for
forall free1 in                           -- arguments var0,var1.
let env0={var0 : free0, var1 : free1} in  -- Bind args into env.
let bc0=((!env0.var0)&&(env0.var1>70)) in -- Branch condition.
let vc0=(bc0->(!env0.var0)) in  vc0       -- VC:not weightOnWheels.
```

During the development of the Ivory-ACL2s interprocedural analyzer, we established a suite of tests to cover the interesting corners of the language. Combining both assertions and procedure contracts, the test suite comprised 61 checks of which 54 were verified automatically by the analyzer. One set of tests of specific interest are Ivory compiler generated assertions, which protect loop bounds, index casting, numerical overflows, and other security impinging aspects of the language. Our limited test suite produced 6 compiler generated assertions, 5 of which were verified automatically. The one that failed verification was bounds checking an index type, though it can be argued this check is not necessary because the index's bounds is enforced by the Ivory type system. Further investigation is warranted to determine verification performance on larger, real-world examples.

4 Formal Methods to Support Application Developers

Both the FORMED and SITAPS projects demonstrate the effective use of DSLs and code generation to make formal methods accessible to developers.

Our approach in both of these projects has been to start with the goal of supporting a developer in creating high assurance software from an environment that includes the normal tools they are used to working with and that also supports a development process that is a consistent superset of the normal process they might use. The development methodology we advocate consists of:

1. Model - capture the software specification in a DSL that can be used to generate code and additional software artifacts
2. Test - perform simple unit testing to ascertain that the model properly captures the most important properties of the specification
3. Plan Proofs - define invariants, pre and post conditions, that are important to obtaining confidence in the implementation
4. Prove - prove properties (only after initial testing indicates proof is likely to succeed) or generate counter-examples. Repair model as needed.
5. Correspondence test - confirm correspondence of model and code and provide traditional visible testing evidence that software meets specification

Correspondence testing is an important aspect of this approach that varies from proof-first development approaches that derive executable code from formal models. The derivation guarantees that the implementation refines the model and thus that proofs at the model level apply to the implementation. This approach, while principled, places the modeling and proof task ahead of the application developer's primary implementation task. It also exposes the development to a common problem that formal methods do not provide an "anytime confidence" approach to development. When proving properties using formal reasoners your confidence is either 0% (unproven) or 100% (proven). The methodology described above provides increasing confidence as more proofs and tests pass and limits the proof effort while focusing on producing an executable software artifact.

5 Related Work and Conclusion

Kestrel Institute's Specware [10] tool synthesizes deployable code from formal specifications by process of successive refinements, with proofs of each refinement step, but requires a proof-first approach. Coq is another, more common, proof first language that has the ability to generate code.

Other efforts have mapped UML to various formal methods languages, such as CSP [1], Z [3], and Alloy [4]. AADL is another example of a graphical language used in both development and formal verification. In [8], the LLVM intermediate language is translated into ACL2 for both testing and low-level theorem proving.

Domain-specific languages enable application development at a higher conceptual level than general purpose languages, but hide their real semantics within the code generator. A shallow embedding of these languages in a formal language like ACL2s enables DSL semantics to be specified, reasoned about, and used to prove properties about applications. The executability of ACL2s also allows it to be used to verify the correct operation of deployable code through test correspondence. Shallow embedding also broadens the user base of formal methods, giving application developers the ability, through the DSL, to create formal models.

References

1. I. Abdelhalim, S. Schneider, and H. Treharne. Towards a practical approach to check UML/fUML models consistency using CSP. In *Formal Methods and Software Engineering*, pages 33–48. Springer, 2011.
2. `http://www.omg.org/spec/ALF`.
3. N. Amálio, S. Stepney, and F. Polack. Formal proof from UML models. In *Formal Methods and Software Engineering*, pages 418–433. Springer, 2004.
4. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: a challenging model transformation. In *Model Driven Engineering Languages and Systems*.
5. H. R. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon. The ACL2 Sedan theorem proving system. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 291–295. Springer, 2011.
6. H. R. Chamarthi, P. C. Dillinger, and P. Manolios. Data definitions in the ACL2 Sedan. In *ACL2 Workshop*, volume 152 of *EPTCS*, pages 27–48, 2014.
7. `http://www.omg.org/spec/FUML`.
8. D. S. Hardin, J. A. Davis, D. A. Greve, and J. R. McClurg. Development of a translator from LLVM to ACL2. volume 152 of *EPTCS*.
9. D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
10. R. Jüllig, Y. Srinivas, and J. Liu. SPECWARE: an advanced environment for the formal development of complex software systems. In *Algebraic Methodology and Software Technology*, pages 551–554. Springer, 1996.
11. `http://www.pathmate.com`.