Slang: The Sireum Programming Language*

Robby[1] and John Hatcliff[1]

Kansas State University, Manhattan KS 66506, USA
{robby,hatcliff}@ksu.edu

Abstract. This paper presents design goals, development approaches, and applications for Slang – a subset of the Scala programming language designed for engineering high assurance safety/security-critical systems. Rationale is given for specializing Scala for Slang so as to retain Scala's synergistic blend of imperative, functional, and object-oriented features while omitting and tailoring features that make formal verification and other analyses difficult. Strategies for enhancing the usability of Slang are discussed including integration with the broader Scala/JVM ecosystem, compilers, and development environments. A number of accompanying Slang tools are described including Slang scripting, meta-programming support, and translators to Javascript and native code that enable support for a wide range of deployment platforms. To support deployment on constrained embedded platforms, the Slang Embedded subset and an accompanying C translator generate efficient implementations that avoid garbage-collection and other aspects that hinder deployment and safety/security assurances. We conclude with a discussion of how our experiences with Slang may provide suggestions for the future of programming and programming language design for engineering critical systems.

1 Introduction

Fueled by accelerating hardware capabilities and algorithmic improvements, formal methods have made significant advances in recent decades and have increasingly gained adoption in industry. For example, there are now powerful SMT2 solvers (*e.g.*, [6,26]) that can scale to realistic problems in large-scale industrial use (*e.g.*, [3]). Groundbreaking and technically challenging work to formally prove correctness of complex hardware and software such as microprocessors (*e.g.*, [35]), OS kernels (*e.g.*, [22]), and compilers (*e.g.*, [25]), among others, have demonstrated the effectiveness of interactive theorem proving.

In our research, we have applied formal method techniques such as software model checking (*e.g.*, [30,16,13]), data/information flow analyses (*e.g.*, [33,34,2,32]), symbolic execution (*e.g.*, [17,7]), interactive and automated theorem proving (*e.g.*, [37,8]), and abstract interpretation (*e.g.*, [37,19]), as well as formal specification languages (*e.g.*, [2,31,14]), on extremally diverse kinds of systems such as real-time component-based embedded systems, concurrent Java, sequential

* Work supported in part by the US Defense Advanced Research Projects Agency, US Air Force Research Lab, US Army, and US Department of Homeland Security.

SPARK/Ada, and mobile Android applications. Many of these recent tools were implemented in earlier generations of our Sireum platform, which is an outgrowth effort of the Bogor [30] model checking framework (which itself is an outgrowth of the Bandera software model checker project [13]). Sireum embraces the Bandera philosophy that usability in formal method tools is a crucial factor for gaining adoption and reducing application efforts. Moreover, it follows the Bogor philosophy of providing general basic building blocks and frameworks that can be customized for specific domains in order to better capitalize on the specific domain-inherent properties for reducing analysis costs.

In our teaching, in recent years, we were tasked to teach manual and automatic program verification for undergraduate sophomores, under the guise of an introductory course on logical foundations of programming that we inherited from David A. Schmidt – a highly-respected researcher and educator in programming languages and formal methods [4]. One might imagine that teaching manual/automatic formal verification to sophomores is a tall order, but Schmidt figured out how to package it effectively. As a teaching aid, he developed a semi-automatic program verifier (which includes its own arithmetic semi-decision procedure) for a "baby" Python language with contract and proof languages that produces an HTML-based verification report.

While it achieved its goals, we desired to have a more modern and automatic tool (while still supporting semi-manual proving as an option) that leverages SMT2 solvers and equipped with a seamless IDE integration that: (a) checks sequent proofs and verifies programs in the background as students type, and (b) asynchronously highlight issues directly in the IDE. This gave rise to the development of the Sireum Logika program verifier and proof checker [39], whose input language is a "baby" Scala script (with contract and proof languages expressed inside a custom Scala string interpolator). Since its initial deployment in the beginning of 2016, Logika has been used to teach around two hundred students each year at K-State. In addition, in the past couple of years, our colleagues at Aarhus University have adopted Logika for teaching logic courses.

Motivated and encouraged by these research and teaching experiences (and inspired by all the great work by others), we desired to consolidate all our efforts. Hence, in the beginning of 2017, we initiated the development of the next generation Sireum platform equipped with its own programming language with supporting specification and proof languages. These inter-connected languages and formal method tools aim to achieve higher levels of safety and security assurances by selectively designing the language features to significantly reduce formal analysis costs, as well as the cost to apply the analyses by making them more usable (*i.e.*, usable enough for teaching undergraduate sophomores).

In this paper, we present the Slang programming language – one result of the multi-year effort described above. Section 2 presents our design guidelines/decisions, which motivated the language features described in Section 3. Section 4 discusses Slang's implementation, and Section 5 describes how Slang have been applied to several domains as parts of our broader validation efforts. Section 6 relates Slang to other work, and Section 7 concludes with some future work.

2 Design

This section describes some of our design guidelines and decisions for Slang.

Usable safety/security first: We aim to provide a language that is amenable to formal verification and analysis. While one route is to emphasize powerful specification logics and verification using manually-oriented proof assistants to handle language complexity (*e.g.*, [10]), we instead choose to carefully engineer the language features and semantics to ease verification. Our (agile) approach is to start "small" and "grow" the language as desired while ensuring that new features do not introduce disproportionate analysis complexity. While we prioritize "verifiability" and "analyzability", we are also mindful of performance and memory footprint. We recognize that language usability is very important, since the ability to achieve safety/security is not worth much if the language is very hard to wield, detrimental to productivity, and impedes explanation of best principles of system engineering, coding, and logic-based program reasoning.

Gradual effort/reward assurance workflow: While we *design* the language for high assurance, we do not always *require* high assurance. Developers should be able to program in the language without being burdened about ensuring, *e.g.*, functional correctness. In these situations, automatic formal methods may be used "behind the scenes": (a) to help developers *understand the behavior of a program* (rather than for verification), and (b) to support other quality assurance techniques (*e.g.*, simulation and testing). As higher assurance levels are required, developers can incrementally employ various formal method techniques/tools.

Rich, high-level language features: As computing technology advances significantly, modern programming languages increasingly include a rich collection of high-level language features to increase productivity. Object-oriented programming language features (*e.g.*, classes and interfaces, dynamic dispatch, *etc.*) are useful for organizing large codebases while providing extensibility. Imperative programming language features are convenient and familiar to most developers. Functional programming language features provide simpler programming mental models (*e.g.*, side-effect free computations, *etc.*) and highly abstract features (*e.g.*, pattern matching, *etc.*). We would like to adopt a mix of these features.

Multi-platform support, from small to large: We want the language to support development of a variety of systems, from small embedded systems (*e.g.*, running inside a micro-controller with 192kb memory) to large applications running on powerful workstations or servers. In addition, we want at least one platform that supports very easy deployment, prototyping, debugging, and simulation (*e.g.*, JVM). This is beneficial for classroom teaching as well as for introducing new researchers to the language and associated formal method tools. Moreover, industrial development can often benefit from: (a) capabilities to rapidly prototype systems on an easy-to-use platform, and (b) facilities that help re-deploy prototyped systems to hardened product platforms.

Integration with existing/legacy systems, libraries, and other languages: While we want our language to be used as often as possible, we need

it to be able to integrate with existing systems and libraries and to provide interoperability with other languages. This allows gradual language adoption or a mixed system development involving different languages, as well as being able to reap benefits from existing efforts. In safety-critical systems, there will inevitably be low-level system code and device drivers that cannot be coded efficiently in a language that emphasizes verifiability and clean abstractions. So there needs to be some way to cleanly interface with such code.

Small-scale development/maintenance: As academics with limited resources and a high turn-over of team members (*i.e.*, students), one pragmatic criterion is that the language infrastructure and tooling should be able to be developed by a very small team (in fact, by a single person so far). This necessitates us to leverage thriving ecosystems instead of building a new one from the ground up.

Based on the above, we decided to realize Slang by leveraging the Scala programming language. Scala offers a good mix of object-oriented, imperative, and functional programming language features. It is built on top of the popular, multi-platform Java Virtual Machine (JVM) ecosystem, which brought about rich tooling support such as dependency management and compilation to native using GraalVM. Scala tooling also offers other compilation targets such as Javascript using Scala.js (and native code using Scala Native).

Despite its many strengths, Scala is a complex language, and thus difficult to highly assure for safety/security. However, its flexible language features (*e.g.*, powerful type inference, macros, *etc.*), open-source/development, and pluggable compiler pipeline architecture allow one to effectively customize the language. This is actually the primary reason why we settled on Scala as no other language tooling offered a similar customization level. Other weaknesses are Scala's slow compilation speed and IDE support issues (not as polished as Java's). All things considered, we deemed Scala's benefits far outweigh its downsides.

Hence, we *specialized* Scala for Slang. We adopted Scala's syntax (*i.e.*, Slang's syntax is a proper subset of Scala's), narrowed down its language features significantly (but still included many powerful features), and tailored its semantics via program transformations implemented in a compiler plugin and Scala macros (thus, enabling Scala/JVM tools to be used for Slang).

3 Features

In this section, we illustrate how Slang design goals are achieved using selected language features. Due to space constraints, we cannot illustrate all of the language features in detail. The Slang website [40] provides a repository of examples and an extensive language reference manual.

Reasoning about object references and heap-allocated data is one of the great challenges of program verification. Many research efforts have tackled this challenge by leaving the language largely unconstrained but then adopting separation logics (*e.g.*, [10]) or advanced type systems (*e.g.*, [12]) to support reasoning. Other efforts such as SPARK/Ada constrain the language to avoid introducing

aliasing and other problems, which allows the specifications and logic to be simpler. We take the latter approach and constrain the language while attempting to preserve many features of Scala's type system and its synergistic fusion of imperative, functional, and object-oriented programming language features.

Distinct mutable and immutable objects: Slang distinguishes between mutable and immutable object structures *statically*. That is, object type mutability is declared as part of type declarations. Immutability is strict in Slang, *i.e.*, immutable objects cannot refer to mutable ones, and violations of this rule are detected by type checking. Mutable objects, however, can refer to immutable ones. The main benefit of using immutable structures is that they simplify reasoning because aliasing becomes a non-issue. On the other hand, mutable objects are often more convenient to transform via destructive updates for many developers.

Sole access path: One of the main difficulties in program reasoning is analyzing mutable object structure in the presence of aliasing. Slang reduces this reasoning complexity by adhering to the following runtime invariant property:

> *At any given program point, a mutable object is observationally reachable only by (at most) one access path.*

One (intended) consequence is that cyclic object structures are disallowed.[1] In general, immutable structures can form directed acyclic graphs while strictly mutable ones can only (at most) form trees.

Adherence to the invariant is facilitated by restricting aliasing on mutable objects. The typical language features that introduce aliasing are assignment (including pattern matching variable binding, for-each loop, *etc.*) and parameter passing. In Slang, assigning a mutable object to a variable (possibly) creates a deep mutable structure copy before updating the variable (if the object has previously been assigned to a variable).

This design choice trades off code performance and better memory utilization for ease of reasoning. In many cases, the decrease in performance/utilization is not detrimental to the coded application. When needed, the code can be rewritten and optimized (justified by empirical data such as profiling and memory analyses – not simply due to the pursuit of premature optimizations). Using an ownership type system is an alternate approach to tame aliasing [12] (*e.g.*, Rust). However, there is increased specification effort in using ownership types, and in languages that adopt them as the *default case*, developers must take on the increased specification burden even when the associated payoffs are small.

We prefer a gradual effort/reward approach in which developers can initially code without such burdens as the default, and only put in additional effort when the situation warrants. This enables developers to focus on code correctness (e.g., established using Slang's contract checking), and then code optimizations can be introduced when needed (and the contract checking framework can be used to

[1] This does not preclude graph algorithms from being written in Slang; in fact, the Slang runtime library provides a graph library (used in [33]), including cycle detection (graphs are realized using indexed node pools with pairs of indices as edges).

prove that the optimized code still satisfies its contract or equivalent to the less optimized version). The implicit copying of mutable structures does sometimes introduce unanticipated behaviors for developers new to Slang due to loss of anticipated data structure sharing. However, tools can be developed to highlight code regions where copying occurs. More approaches can be added to Slang in the future to further facilitate performance improvement/memory usage reduction, but they should not always be required if additional specifications are needed.

This leaves us with aliasing via parameter passing, which Slang allows with restrictions. Specifically, a method invocation sufficiently maintains the sole access path invariant when the mutable object structures being passed (and accessed "global", instance, or closure-captured variables) are disjoint/separate, and this can be established by ensuring that no two abstract access paths to mutable objects involved in the method invocation are a prefix of one to another. By abstract, we meant an access path p of the form: (1) variable reference x, (2) field access $p.x$, or (3) sequence indexing $p(\top)$, where the index value is abstracted away as $\top$. For example, consider a method invocation $p_0.m(p_1,\ldots,p_n)$ where each $p_{i\in\{0,\ldots,n\}}$ is an abstract access path to a mutable structure, and where `m` accesses enclosing variables of mutable type represented as abstract paths $p_{n+1},\ldots,p_{n+m}$ (which can be inferred [32]). The method invocation maintains the sole access path invariant if: $\neg\exists\ i,j \in \{0,\ldots,n+m\}.\ i \neq j\ \wedge\ \mathit{prefix}(p_i,p_j)$. A more precise condition can be used if one is willing to prove all index values of same abstract path prefixes are all distinct at the invocation program point.

One useful Slang programming pattern to reduce implicit copying involves exploiting Slang's restricted aliasing by parameter passing. Instead of returning mutable objects as method results, which may trigger copying when returned objects are assigned to variables, mutable objects can be passed down the call chain to store computed results via destructive updates. This approach, which we nicknamed as the Slang "hand-me-down" maneuver, works very well even for programming small embedded systems (*i.e.*, to avoid dynamic memory allocation efficiently), but at the cost of what one might consider as "inelegant" code.

In short, Slang's approach to aliasing focuses the reasoning/verification concerns to statically-defined classes of objects (*i.e.*, mutable) at a single well-defined and easily-identifiable program construct (*i.e.*, method invocation), which is also the main location of concern in compositional program verification and formal analyses. Hence, addressing aliasing can go hand-in-hand with other compositional assurance approaches such as for ensuring absence of runtime errors (*e.g.*, buffer overflows), functional correctness, and secure information flow.

Type system: Slang does not have a notion of a "top" type like `Object` in Java or `Any` in Scala for more meaningful sub-typing relationships in program reasoning. Moreover, Slang does away with the problematic `null` value in favor of optional types.[2] There is no implicit type coercion in Slang as such coercion may be unintended and non-trivially affect program reasoning. Object identity is non-

[2] Memory footprint optimizations in the Scala compiler plugin for Slang include flattening `None` into `null` and `Some` to its contained value internally for object fields, but optional values are used for field accesses.

observable (one can always add a distinguishing field value when needed), and object equality tests are structural equivalences by default (*i.e.*, equality tests can be customized). Generics are supported, though currently unconstrained. In the future, type constraints over generics such as "sub-type of T" might be added, but as mentioned previously, we wanted to start "small".

Slang (immutable) built-in types are: `B` (boolean), `Z` (arbitrary-precision integer), `C` (character), `String`, `R` (arbitrary-precision decimal number), `F32` (32-bit floating-point number), `F64` (64-bit floating-point number), and `ST` (a template engine implemented using a custom Scala string interpolator).

Developers can introduce two kinds of custom integer types: (1) range integer types, and (2) bit-vector integer types. Range integer types can optionally specify min/max values (*i.e.*, they can be arbitrary-precision). Operations on range integers are checked that they stay within the specified min/max values. Range integer types do not provide bit-wise operations, which are offered by bit-vector integers. However, operations on bit-vector integers are not range checked, and they are backed and operated (in the generated code) using either 8-bit, 16-bit, 32-bit, and 64-bit values, using the smallest that fits. The reason why Slang distinguishes range and bit-vector integer types is because automated analyses on the latter are often significantly more expensive. That is, developers should use range integer types over bit-vector if bit-wise operations are not needed. Below are some examples of range and bit-vector integer type declarations.

```
@range(min = 0) class N                 // natural number (arbitrary-precision)
@range(min = 1, max = 10) class OneToTen // 1 .. 10 range int
@bits(signed = T, width = 8) class S8    // 8-bit signed bit-vector int
@bits(min = 0, max = 3) class U2         // 0 .. 3 unsigned bit-vector int
```

The Slang compiler plugin automatically generates supporting operations on such types (*e.g.*, addition, subtraction, *etc.*) as the (Scala value) class methods.

Slang built-in sequence (array) types are immutable `IS[`I`, `T`]` and mutable `MS[`I`, `T`]`, which are parameterized by index type I and element type T. The index type I has to be either `Z`, a range integer type, or a bit-vector integer type. A sequence's size is fixed after creation, and accesses on sequences are checked against the sequence index range. Append, prepend, and remove operations on `IS` and `MS` are provided, and they always create a new sequence. Tuple types `(`T_1`, ..., `T_N`)` are supported and their mutability is determined based on the contained element types. For (higher-order) closures, function types `(`T_1`, ..., `T_N`) => `U are also supported and classified based on their "purity" (discussed below). (When using (higher-order) closures in mixed paradigm languages like Slang, high assurance applications should use pure closures at present, if at all, as contract languages/analyses on "impure" closures are still subjects of our ongoing research.)

Slang interfaces (traits) and classes are distinguished by their mutability (and "sealing") as illustrated by the following example.

```
@sig trait I { ... }            // immutable interface
@datatype trait ID { ... }      // immutable sealed interface
@datatype class D(...) { ... }  // immutable final class
@msig trait M { ... }           // mutable interface
@record trait MR { ... }        // mutable sealed interface
@record class R(...) { ... }    // mutable final class
```

Traits can only define/declare methods (with/without implementations), but fields are disallowed. Traits can extend other traits, and classes can implement traits. Implementations of sealed traits have to be defined in the same file.

To keep it simple, classes are always final, hence, there is no class inheritance in Slang, only trait implementations. Inherited methods can be overriden. Method definitions that are inherited by a trait or a class T with the same name (and similar signatures) originating from different traits have to be overriden in T, so developers are explicitly made aware of a potential reasoning issue.

Classes can only have one constructor, and constructor parameters are fields. Fields can have read access (`val`, which is also the default) or read/write access (`var`), and both can refer to mutable and immutable objects. A `@datatype` class, however, can only have `val`s as fields, while a `@record` class can also have `var`s. All fields and methods have to be explicitly typed, which simplifies type inference and eases parallel type checking (after type outlining). All traits and classes regardless of mutability can have methods with side-effects (impure).

Slang `object` O `{ ... }` definitions are singletons like in Scala, except that they cannot implement traits and cannot be assigned to a variable or passed around. Therefore, they simply serve to group "global" fields and methods, and consequently, they are allowed to refer to mutable structure or define `var`s.

Method purity: Slang classifies methods based on the kinds of side-effects that they might make: (a) impure, (b) `@pure`, (c) `@memoize`, and (d) `@strictpure`. Impure methods are allowed to (destructively) update objects without restrictions. Methods (and functions) annotated with `@pure` cannot update existing objects but they can create and update the newly created objects as part of their computation. They can also define local `val`s and `var`s, use loops, and recurse. They cannot access any enclosing `val` which holds a mutable object or any enclosing `var` during their execution. `@memoize` is `@pure` with (non-observable) automatic caching. Hence, `@memoize` (and `@pure`) are observationally pure [28]. `@strictpure` methods have further restrictions; they cannot update any object, and they cannot define local `var`s (only `val`s) or use loops, but can recurse.

Pure methods simplify reasoning and are useful for specification purposes. We distinguished `@strictpure` methods since they can be directly translated to logical representations, and therefore they do not require explicit functional behavior contract specifications for compositional verification purposes. Thus, they can be treated directly as specifications (after checking for runtime errors).

Control structures: Slang supports Scala's if-conditional and while-loop, as well as for-loop and for-yield-comprehension, including forms with multiple iterators and conditions. Slang fully embraces Scala's pattern matching constructs with only minor constraints (related to loss of information due to erasure). Exceptions and exception handling are not supported as they complicate reasoning.

To simplify formal analyses (and improve readability), Slang restricts other Scala syntactic forms. For example, a code block that evaluates to a value expressed by e – $\{\ldots; e\}$, is not part of Slang's expression language. Slang categorizes a special syntactic form ae that is only allowed as an assignment's right-hand side or as a function closure definition body (liberally) defined as:
$ae ::=$ e | $\{\ldots; ae\}$ | `if (`e`)` ae `else` ae | `return` e | e `match { `*case*`+ }`
where each *case* pattern matching body evaluates to a value (`return` is disallowed in closure definitions). Note that an if-conditional without a code block (similar to Java's ternary `?:` operator) is part of Slang's expression language.

Extensions: As described above, many Scala features are excluded, partly because we wanted to simplify reasoning and partly because we wanted to start "small" in the language design. One additional noteworthy exclusion is concurrency features. This is in part due to the fact that in our current industrial research projects, we support concurrency using automatically generated real-time tasking structures derived from AADL [21] architecture models using our HAMR framework [18] described in Section 5. In the AADL computational model, component implementations (e.g., as written in Slang or C) are strictly sequential, implementing functions that transform values on component input ports to values on component output ports. Task scheduling, communication, and resolving contention on component ports is handled by the underlying run-time middleware. This organization of computation and tasks enables contract-based reasoning at the architectural level. Alternatively, there is a promising concurrency approach currently being incubated in Project Loom [38] – Java Fibers, which are lightweight and efficient threads powered by continuations implemented in the JVM, which seems worth waiting for. Regardless, Slang provides extension methods (akin to Bogor extensions [30]) to extend its capabilities by leveraging existing libraries as well as providing interoperability with other languages. Below is an example extension that provides a (pure) parallel map operation, which is heavily used in the Slang compilation toolchain described in Section 4.

```
@ext object ISZOpsUtil {
  @pure def parMap[V, U](s: IS[Z, V], f: V => U @pure): IS[Z, U] = $
}
```

For JVM, Scala.js Javascript, and GraalVM native compilation targets, the Slang compiler plugin rewrites `$` as method invocation `ISZOpsUtil_Ext.parMap(s, f)`, whose implementation is written in Scala using its parallel collection library for the JVM and native targets, and using a sequential map operation for the Javascript target. The `@ext` annotation can optionally accept a string of the target `object` name that provides the method implementation as an alternative of using the extension object identifier with the `_Ext` suffix convention. In general, the implementation can be written in any language of the hosting platform, and from this implementation one can call any library available on the platform.

One issue with extensions is that for Slang analysis/verification to be sound, an extension implementation has to adhere to Slang's sole access path invariant and obey any stated contract for the extension. However, conformance to these requirements cannot generally be established from within Slang or by using Slang

tools. Thus, other assurance approaches have to be used, or conformance has to be considered as an undischarged assumption.

Language subsets: We recognize that not all Slang features are desirable for a certain application domain. For example, when targeting high assurance embedded systems, closures and recursive types are undesirable due to the dynamic memory management requirement for realizing such features. Thus, it is often handy to be able to create subsets of Slang for particular domains. These subsets can be enforced by a static analysis over fully resolved Slang abstract syntax trees. We have created a subset of Slang called *Slang Embedded* that is targeted for translation to efficient embedded C (see Section 4).

Specification and proof language support: As Slang is also an input language for the next generation Logika tool that we are currently developing, as well as for other formal analyses that we plan to develop, one of its design goals is to facilitate (possibly expanding) families of specification and proof languages.

As previously mentioned, we would like Slang's syntax to be a proper subset of Scala's syntax, because this allows existing Scala tooling (*e.g.*, IDEs) to work without much modification, thus lowering our tool development overhead. Drawing from an earlier work [29] that uses Java method invocations to represent contract forms, we also found that we could nicely simulate grammars using Scala's method invocation and special syntax features (internal DSLs). Below is an example behavioral contract of a method that `swap`s two sequence elements.

```
def swap[I, T](s: MS[I, T], i: I, j: I): Unit = {
  Contract(
    Reads(),           // read accesses (by default, parameters are included)
    Requires(s.isInBound(i), s.isInBound(j)),              // pre-conditions
    Modifies(s),                                           // frame-conditions
    Ensures(                                               // post-conditions
      s(i) == In(s)(j), s(j) == In(s)(i), s.size == In(s).size,
      All(s.indices)(k => (k != i & k != j) ->: (s(k) == In(s)(k)))))
  val t = s(i); s(i) = s(j); s(j) = t
}
```

`Contract`, `Reads`, `Requires`, `Modifies`, `Ensures`, `All`, and `In` are Scala methods:

```
object Contract {
  def apply(arg0: Reads, arg1: Requires, arg2: Modifies, arg3: Ensures): Unit =
    macro Macro.lUnit4
}
object All {
  def apply[I, T](seq: IS[I, T])(p: T => Boolean): B = { /* ... elided */ }
}
def Reads(accesses: Any*): Contract.Reads = ??? // throws NotImplementedError
def Requires(claims: B*): Contract.Requires = ???
def Modifies(accesses: Any*): Contract.Modifies = ???
def Ensures(claims: B*): Contract.Ensures = ???
def In[T](v: T): T = ??? // retrieve v's pre-state; usable only inside Ensures
```

`Contract` is realized using a Scala macro that effectively erases any invocation to it, thus it does not affect code runtime behaviors.[3] All other methods are simply stubs with *types that enforce grammar rules*. Below is another example

[3] A runtime contract checker (similar to [36]) can be developed in the future for testing purposes (or for contract enforcement with various mitigation options).

illustrating one of Slang theorem forms that proves a well-known syllogism using Slang's proof language, which also exploits Scala's syntactic flexibility:

```
@pure def s[U](human: U => B@pure, mortal: U => B@pure, Socrates: U): Unit = {
  Contract(Requires(All{(x: U) => human(x) ->: mortal(x)}, human(Socrates)),
           Ensures (mortal(Socrates)))
  Deduce(1 #> All{(x: U) => human(x) ->: mortal(x)}      by Premise,
         2 #> human(Socrates)                            by Premise,
         3 #> (human(Socrates) ->: mortal(Socrates))     by AllElim[U](1),
         4 #> mortal(Socrates)                           by ImplyElim(3, 2))
}
```

`Deduce`, like `Contract`, is erased in compilation. It serves to enumerate proof steps. Each step: (a) requires an explicit claim (for proof readability and auditability); (b) is uniquely numbered/identified in that particular proof context; and (c) has to be justified `by` using some proof tactics implemented as Logika plugins (*e.g.*, `Premise`) or applying lemmas/theorems (*e.g.*, `AllElim`, `ImplyElim`). (Details of these mechanisms will be provided in forthcoming documentation for the Logika next generation tool.)

`Contract`, `Deduce`, and other specification and proof language constructs described above are specially recognized by the Slang parser (described in the next section) and they are treated as first-class Slang constructs in Slang abstract syntax tree representations and downstream Slang compiler phases and toolchains. Hence, Slang has to be updated to support additional specification and proof constructs as more formal method tools for Slang are introduced or enhanced.

One disadvantage of this approach is that the specification and proof languages that can be introduced in Slang are limited by Scala's expressive power. However, we have found existing Scala tool support works well. For example, IntelliJ's Scala support for code folding, refactoring (*e.g.*, variable renaming), hyperlinking, type checking, *etc.*, works for Slang's specification and proof languages. This enables modern software engineering tool capabilities to be applied to the *specification and proof engineering* process.

4 Implementation

Slang compilation was first implemented using the Scala compiler extended with a custom compiler plugin and supporting macros. This provided compilation to JVM, Javascript, and native executables almost for free. With this approach, we prototyped Slang features rapidly (including IntelliJ integration using a custom plugin described later in this section). Because the Slang runtime library is written in Slang itself (aside from built-in types), this pipeline also continuously tested and validated Slang language features. After the prototype relatively stabilized (which took around half a year), we began a more ambitious validation by implementing the Slang front-end mostly using Slang itself and followed this by implementing, in Slang, a C back-end appropriate for embedded systems. This section describes the resulting compiler pipeline and Slang IntelliJ integration, as well as a supporting command-line build tool we recently developed.

The implementation guideline that we follow in our platform engineering effort is to think "BIG". That is, we want to take advantage of the availability

of powerful machines with multi/many-cores and high memory capacity to gain reduction in the most precious resource of it all, *i.e.*, time.

Front-end: Since Slang is based on Scala's rich and featureful syntax, building the parser itself was an early challenge. We ended up using the Scala parser from the open-source `Scalameta` library to produce parse trees, which are translated to Slang-based abstract syntax trees (ASTs). This is the only front-end part that is not written in Slang.[4] Another issue is how to (speedily) distinguish Slang vs. Scala programs during parsing. We resolved this by adopting the convention that Slang files should have `#Sireum` in its first line (in a comment).

Given this approach, the details of the front-end can be summarized as follows (assuming program well-formed-ness for presentation simplicity). The front-end accepts an input list of source paths which are mined for `.scala` files with `#Sireum` in its first line; found files are then read into memory. Each file's content is then (optionally `parMap`) parsed to produce Slang ASTs that are then processed to create file symbol tables. The file symbol tables are reduced into an overall symbol table, and programs are then type outlined. Type outlining processes type, field, method signatures (which, as we previously mentioned, have to be explicitly typed), and contracts, without processing field initializations and method bodies. This produces a transformed symbol table, ASTs, and a type hierarchy. Finally, each trait, class, and `object` AST is (`parMap`) type checked/inferred and reduced to produce yet another transformed symbol table and set of ASTs (type hierarchy is unaffected). All Slang symbol table, AST, and type hierarchy objects are implemented as immutable structures to enable safe (parallel) transformations. However, error, warning, and informational message reporting is done using a mutable `Reporter` that accumulates (immutable) messages as it is passed through the compilation routines using the "hand-me-down" maneuver (recall that `@pure` can create and mutate new objects).

Embedded C Back-end: We now summarize issues specific to the *Slang Embedded* subset of Slang and its translation to C. Since we target high assurance embedded systems, we decided that the generated C code should not dynamically allocate memory (a common restriction in safety-critical systems). This avoids runtime garbage collection, which helps ensure predictable behaviors.[5] Therefore, when using the Slang Embedded subset, all memory allocations must be either globally-allocated statically or stack-allocated. One consequence is that all sequences have to be bounded (the bounds are user-configurable). In addition, language features that require dynamic allocation such as recursive types (*e.g.*, linked-lists, *etc.*) and closures are prohibited. Recursion is allowed, but it is best if tail recursion is used (otherwise, call chain depths should be estimated/bounded). In summary, the features available in the Slang Embedded subset

[4] Aside from extensions in the Slang runtime library for file access (and spawning processes, OS detection, *etc.*), which are available on JVM and native targets.

[5] We initially planned to offer C compilation with garbage collection, but GraalVM or Scala Native can be used instead. We may reconsider such approach in the future.

align with what one would typically use when programming embedded software. Best practices include using statically-allocated bounded object pools if needed.

We want the generated C code to be compilable by the popular `gcc` and `clang` compilers. We also would like to leverage the great work on the CompCert verified C compiler[25], which provides strong guarantees that produced binary code faithfully reflect the semantics of its sources (this enables a possible certified translation from Slang to CompCert C sometime in the future). Thus, we ensure that the generated C code conforms to the C99 definition, which is supported by all three compilers above. We treat any warning from the C compilers as a bug in the Slang-to-C translation. In addition, we want the generated C code structure to match the structure of the corresponding Slang code to help enable traceability and to ease debugging/maintenance of the translator. Moreover, we want the generated C code to maintain Slang-level debugging stack information. For example, whenever there is an assertion error, one should get a similar stack trace to what is provided by the JVM, including Slang source filename and line number information. Finally, we would like to be able to edit, test, debug, and integrate the generated C code, so the translator should generate supporting files for use in an IDE. We decided to support CLion – IntelliJ's sister commercial product that accepts CMake configurations as project definitions.

To facilitate the above, the generated C code makes heavy use of C macros to ensure that generated code is similar in structure and appearance to the source. The macros also help maintain Slang-level source information (which can be optionally turned off). For example, the following two Slang methods:

```
def foo(): Unit = { println("foo") };    def bar(): Unit = { foo() }
```

are translated as follows (edited for brevity; macro expansions in comments):

```
typedef struct StackFrame *StackFrame;
struct StackFrame { StackFrame caller; /* ... */; int line; };

Unit foo(STACK_FRAME_ONLY) {            // ...(StackFrame caller)
  DeclNewStackFrame(caller, /* ... */, // struct StackFrame sf[1] = ...
                    "foo", 0);          //        .name = "foo", .line = 0 } };
  #ifndef SIREUM_NO_PRINT
  sfUpdateLoc(1);                       // sf->line = 1;
  { String_cprint(string("foo"), T);
    cprintln(T);
    cflush(T); }
  #endif
}

Unit bar(STACK_FRAME_ONLY) {            // ...(StackFrame caller)
  DeclNewStackFrame(caller, /* ... */, // struct StackFrame sf[1] = ...;
                    "bar", 0);          //        .name = "bar", .line = 0 } };
  sfUpdateLoc(1);                       // sf->line = 1;
  { foo(SF_LAST); }                     // foo(sf);
}
```

As illustrated above, each function stack-allocates `StackFrame` to store source information, which also maintains a pointer to the caller's stack frame. Source location information is updated for each corresponding Slang statement using `sfUpdateLoc`. Thus, at any given C program point, the Slang call stack information is available for, *e.g.*, printing to console (or when debugging in CLion). The `StackFrame` stack allocation strategy is the same that is used to stack allocate

Slang objects. Each Slang statement may be translated into several C statements as can be observed for the translation of `println` in `foo`. The C statements are grouped inside curly braces for structuring purposes, but more importantly, the grouping serves as a hint to the C compiler that the memory associated with stack-allocated objects inside the curlies can be safely reused after the statement has finished executing, thus reducing memory footprint. Console printing is guarded by a macro, allowing it to be disabled. This is handy when running the code in a small micro controller without a console; that is, the program can have console logging code for debugging purposes, but this functionality can be turned off when deployed to a system without a console.

There are some translation design choices related to namespaces, generics, type hierarchy, and dynamic dispatch. For namespaces, with the exception of Slang built-in types, we use underscore-encoded fully qualified names as C identifiers. This may yield long identifiers, but the approach is predictable and systematic. The translation specializes generics to the specific type instantiations used in the source program. For example, different code blocks will be generated for `Option[Z]` and `Option[IS[Z, Z]]`. This potentially generates a lot of code, incurring memory overhead. We reduce the problem by only translating methods reachable from a given set of entry points. The type hierarchy is realized by translating each trait as a `C union` of its direct subtypes and each class as a `C struct`; the `union`s and `struct`s store an enumeration representing the object runtime type. Dynamic dispatch is handled by generating bridge code for each unique virtual invocation method target that switches on the runtime type stored in the translated `union`/`struct` and calls the corresponding implementation.

Given a set of program entry points, the C back-end first computes a whole-program Slang method call-graph. The type specializer uses the call-graph information to instantiate generic types and methods and specialize them. Finally, `ST` template-based translation processes each type, global field, and method and groups the output code based on the fully qualified name of types and `object`s to which the fields and methods belong. The results are then written to the disk.

The translation executes quickly, so we did not parallelize it. Moreover, unlike the front-end which we developed in a more "functional programming style" where new objects are created as part of the staged transformations, we purposely programmed the C template-based translation in a "more imperative style" that destructively updates template collections to add instantiated templates. We used Slang imperative features to see if there are inherent performance bottlenecks and/or memory consumption issues associated with these imperative features, and we did not experience any.

In our experience using Slang to code both the front-end and the C back-end, we found that mutating objects is convenient but best done locally with shallow mutable structures (which may hold deep immutable structures) to ease reasoning and code maintenance. We believe these are Slang *general* best practices:

"*Immutable globally and deeply. Mutable locally and shallowly.*"

IntelliJ integration and build tool: We strongly believe that IDE support is a crucial productivity enhancing tool that a language should be equipped with. IntelliJ is one of the best Scala IDEs available today. One nice feature is that IntelliJ's Scala type checking is both seamless and perceivably fast when running on modern hardware. IntelliJ's front-end runs in the background as one edits the code and asynchronously highlights issues as they are found. This speeds up development cycles considerably, thus mitigating Scala's slow compilation speed.

IntelliJ's Scala front-end is different than Scala standard compiler's. One issue is that it does not accept Scala compiler plugins and Scala macro support is limited. This means that it does not process Slang program transformations implemented in the Scala compiler plugin for Slang. For example, the `@range`/`@bits` classes do not have supporting operations from the perspective of IntelliJ's front-end as the operations are generated by the Slang compiler plugin.

Fortunately, Scala compiler plugins and macros are widely used in the Scala community, thus, IntelliJ offers injection extension points so compiler plugin developers can provide custom code transformations. Hence, we provide an IntelliJ plugin that implements the injection extension points to introduce Slang program transformations to IntelliJ's front-end. We also limit Slang macros to forms that works well with IntelliJ (*i.e.*, Scala "blackbox" macros). While it took some effort to integrate with IntelliJ, the gained productivity level is far more than enough to make up for it. IntelliJ's injection extension points are very stable so far, meaning that our Slang IntelliJ injection plugin has required little maintenance over time in spite of IntelliJ upgrades.

Another integration issue is configuring IntelliJ for Slang projects. In the past, we have used popular Scala build tools (*e.g.*, sbt and Mill), whose configurations can be imported by/exported to IntelliJ. However, they require some Scala finesse to use effectively. Thus, we developed a build tool for Slang, called Proyek (also written in Slang), whose module configurations are compositional and expressed via Slang `@datatype` class instances in Slang scripts (discussed in Section 5). Proyek can export its configuration to IntelliJ's, and it configures IntelliJ specifically for development using Slang. This enables IntelliJ's built-in build system to be used for compilation, testing, and debugging inside IntelliJ instead of running a separate third-party build tool for these tasks. In addition, Proyek provides command-line tasks for cross-compiling (module-incremental/parallel, including Scala and Java sources), testing, assembling, and publishing.

The above integration works well for using IntelliJ as an IDE for development using Slang. However, the approach by itself does not integrate Slang tools such as the Slang front-end (that checks for, e.g., Slang restrictions of Scala) and formal verification and analysis tools such as Logika. Thus, we recently developed another IntelliJ plugin as a client to a Sireum server that offers Slang tools as micro-services running locally (or, in the future, remotely). These integrations turn IntelliJ as an Integrated Verification Environment (IVE) for Slang – Sireum IVE (in the spirit of [11]), where formal method tools are parts of developers' toolbox for quality assurance that complement, for example, testing.

5 Applications

The previous section described how Slang was validated in part by using Slang to develop the Slang tooling itself. This section summarizes other significant applications that serve as further validation.

Sireum HAMR [18]**:** provides a multi-platform **H**igh **A**ssurance **M**odel-based **R**apid engineering framework for embedded systems. Developers specify component-based system architectures using the SAE standard AADL architecture description language. From an architecture specification, HAMR generates real-time tasking and communication infrastructure code. Developers complete the system by implementing the "business logic" for the components.

HAMR works by first generating a Slang-based intermediate representation of the AADL model (AIR) from the Eclipse-based AADL OSATE IDE which is then used as input for auto-generating the tasking and communication middleware in Slang Embedded. If developers use Slang Embedded to implement component business logic, the completed system can be run on the JVM for prototyping, simulation, and testing. The Slang Embedded system code can be translated to C (using the Slang Embedded C translator), which in turn can be compiled to a native application for macOS, Linux, or Windows. Additionally, a build can be created to run on the seL4 [22] verified micro-kernel to provide guaranteed spatial separation of tasks, which is useful for safety/security-critical systems. HAMR supports an alternate C-based workflow in which developers implement component business logic directly in C, while Slang is used by HAMR "behind the scenes" to generate the C tasking and communication middleware.

In summary, HAMR provides validation of Slang in a number of dimensions; it is used: (a) in meta-programming to describe the structure of models and templates for generating code, (b) to implement the AADL run-time libraries for tasking and communication, (c) to code component application logic for deployment on the JVM, and (d) as translation source for C or native-based applications deployed on multiple platforms.

Web applications: Rich web applications have surged in popularity in recent years; even some desktop applications (*e.g.*, Visual Studio Code) are now developed using web technologies. Using the Scala.js translator from Scala to Javascript, one can code aspects of web applications in Slang. We have found this convenient for building several Sireum tools. It is particularly effective for reporting and visualizing the results of Sireum analyses, because reports, highly interactive analysis visualizations, and even the analysis algorithms themselves can be presented/executed using existing web frameworks, and only a web browser is needed for viewing/executing. This has proved useful in our industrial interactions – engineers can view rich results of example analyses without having to install the full Sireum toolchain. For example, we used the strategy to provide AADL information flow analysis in the Sireum Awas tool [33]. Awas enables users to view and navigate the overall AADL system hierarchically, and to launch and interact with Awas information flow analyses that dynamically highlight how information flows throughout the system. While the Awas web application is

mostly implemented in Scala (as part of the previous generation of Sireum), its core graph representation and algorithms are from the Slang runtime library, which are translated to Javascript via Scala.js.

Meta-programming: Slang has been used to implement several meta-programming tools that use Slang traits/classes as a data schema language. In one tool, Slang-based schemas are processed to generate Slang code to de/serialize the trait and class instances from/to JSON or MessagePack. In addition to AIR, this is used on fully resolved Slang AST, for example, as part of the Slang front-end regression suite that checks the property $AST = deserialize(serialize(AST))$ on the entire Slang front-end codebase itself. In a second tool, code is generated to perform pre/post-order visits/rewrites of the trait and class instances. This is used to implement the C back-end generic type specializer that transforms Slang AST objects into their specialized versions before translating them to C.

Scripting: Slang scripts leverage the Scala scripting facility, and they provide a lightweight way to utilize Slang code and associated formal methods tools. Scripts are ideal for teaching, *e.g.*, program reasoning, as mentioned in Section 1.

In addition to the conventional lightweight coding functionality provided by common scripting languages, Slang scripts can be used to capture contract specification and proof artifacts exclusively, even without executable code present in the script. This provides a convenient high-level specification/proof scripting language instead of having to interact directly with underlying SMT2 tools and theorem provers. For example, when verifying AADL interface contracts between two components in the HAMR framework above, verification conditions are expressed in a Slang script (which takes advantage of Slang high-level types and language features). The script can then be discharged by using Logika.

Scripting is also convenient when embedding domain specific languages (DSLs) in Slang. For example, we defined an embedded DSL for `bitcodec`, which takes a bit-precise "wire" message format specification in a script and generates Slang Embedded message encoder and decoder, for use in implementing communication between HAMR components when processing raw data bitstreams.

We also provide a Slang script variant, called Slash, that can be directly called from both POSIX `sh`-compatible and Windows `cmd` environments, which we heavily use as a means for universally shell scripting complex tasks. Proyek configurations are Slash scripts, which can optionally print module dependencies in GraphViz's dot for visualization purposes. We also use Slash scripts to ease Sireum installation and continuous integration testing on multiple platforms.

6 Related Work

Slang's design is inspired by our earlier work on SPARK in which we developed a symbolic execution-based framework for contract checking [2,7,20], information flow analyses [1,32], and a mechanized semantics in Coq [37]. In this work, we observed first-hand how language feature simplication significantly reduces formal analysis costs. We used the SPARK 2014 semantics concepts in [37] as the

basis for Slang's approach to structure copying/updates, but also significantly extended the scope to address the modern features offered by Scala for small to large application development, with/without runtime garbage collection.

Slang differs from other work that adds custom contract specification notations to an existing language (*e.g.*, JML [23], Spec# [5], Frama-C [15], Stainless/Leon [9], VST [10]), in that Slang specializes/subsets the programming language (Scala) primarily to ease formal analyses. Thus, we have (so far) avoided becoming "bogged down" in complexities in both specification notations and verification algorithms due to having to treat complex language features that impede verification.

In contrast to other work on contract verification that develops an entirely new language specifically for verification purposes (*e.g.*, Dafny [24], Lean [27]), Slang leverages an existing multi-paradigm language with rich tooling support and integration of both Scala and Java ecosystems (*e.g.*, compilers, IDEs). This additional support and ecosystems have enabled us to use Slang to implement significant infrastructure (including Slang and many aspects of the Sireum tools that are bootstrapped and implemented in Slang) and industrial-strength tools such as the HAMR model-driven development framework [18]. This type of large-scale software development and tool engineering would be very difficult to achieve in a programming language that did not integrate with a larger development ecosystem.

7 Conclusion

In this paper, we presented the goals, development approach, and example applications of the Sireum programming language – Slang. With respect to the goals of this ISOLA Track "Programming: What is Next?", we believe our work with Slang explores several important issues related to the future of programming and programming language design described below.

- Increasing numbers of systems need high-assurance for safety and security. What are possible approaches that enable highly automated verification techniques for modern languages with rich features sets? For Slang, we argue that a language can be subsetted for particular domains (eliminating complex features unnecessary for that domain). Base languages that provide powerful syntactic frameworks, customizable compilers, and extensible tooling make it easier to develop usable development environments for such subsets. Scala is one of the few languages to provide these multiple customization dimensions. Thus, while language subsetting for safety-criticality and formal verification is not a new idea, we believe our work is providing additional insights into enabling technologies and specialized subset development approaches.
- Despite tremendous advances in formal methods, there is still a need for increased usability and better integration of specification and verification in developer workflows. Although Slang contracts and verification is not the central subject of this paper, Slang illustrates what we believe are realistic and effective approaches for developer-friendly formal methods integration.

– Regarding development for embedded systems, we believe the approach we are taking with Slang (executable on the JVM for prototyping with rich Scala/Java ecosystems, yet compilable to efficient embedded C code) is relatively unique and exposes possible directions for increasing flexibility and usability of embedded system languages that also utilize modern language features and type systems to avoid code vulnerability and flaws.

Slang is still in active development, though its focus has now shifted to specification and proof languages to support formal analyses. The Slang website [40] hosts a growing collection of language reference, example, and pedagogical resources.

References

1. Amtoft, T., Dodds, J., Zhang, Z., Appel, A.W., Beringer, L., Hatcliff, J., Ou, X., Cousino, A.: A certificate infrastructure for machine-checked proofs of conditional information flow. In: 1st International Conference Principles of Security and Trust (POST). LNCS, vol. 7215, pp. 369–389 (2012)
2. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., Greve, D.A.: Specification and checking of software contracts for conditional information flow. In: Hardin, D.S. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 341–379. Springer (2010)
3. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018)
4. Banerjee, A., Danvy, O., Doh, K., Hatcliff, J. (eds.): Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt [on occasion of his 60th birthday], EPTCS, vol. 129. OPA (September 2013)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS). LNCS, vol. 3362, pp. 49–69 (2004)
6. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: 23rd International Conference Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 171–177 (2011)
7. Belt, J., Hatcliff, J., Robby, Chalin, P., Hardin, D., Deng, X.: Bakar Kiasan: Flexible contract checking for critical systems using symbolic execution. In: 3rd NASA Formal Methods Symposium (NFM). LNCS, vol. 6617, pp. 58–72 (2011)
8. Belt, J., Robby, Deng, X.: Sireum/Topi LDP: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In: 7th joint European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE). pp. 355–364 (2009)
9. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the Leon verification system: verification by translation to recursive functions. In: 4th Workshop on Scala. pp. 1:1–1:10 (2013)
10. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1-4), 367–422 (2018)
11. Chalin, P., Robby, James, P.R., Lee, J., Karabotsos, G.: Towards an industrial grade IVE for Java and next generation research platform for JML. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(6), 429–446 (2010)

12. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, LNCS, vol. 7850, pp. 15–58. Springer (2013)
13. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: 22nd International Conference on Software Engineering (ICSE). pp. 439–448 (2000)
14. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: the Bandera Specification Language. Int. J. Softw. Tools Technol. Transf. (STTT) **4**(1), 34–56 (2002)
15. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - A software analysis perspective. In: 10th Software Engineering and Formal Methods (SEFM). LNCS, vol. 7504, pp. 233–247 (2012)
16. Deng, X., Dwyer, M.B., Hatcliff, J., Jung, G., Robby, Singh, G.: Model-checking middleware-based event-driven real-time embedded software. In: 1st International Symposium Formal Methods for Components and Objects (FMCO). LNCS, vol. 2852, pp. 154–181 (2002)
17. Deng, X., Lee, J., Robby: Efficient and formal generalized symbolic execution. Autom. Softw. Eng. (ASE) **19**(3), 233–301 (2012)
18. Hatcliff, J., Belt, J., Robby, Carpenter, T.: HAMR: An AADL multi-platform code generation toolset. In: 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA) (2021), (to appear)
19. Hatcliff, J., Dwyer, M.B., Pasareanu, C.S., Robby: Foundations of the Bandera abstraction tools. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]. LNCS, vol. 2566, pp. 172–203 (2002)
20. Hatcliff, J., Robby, Chalin, P., Belt, J.: Explicating symbolic execution (xSymExe): an evidence-based verification framework. In: 35th International Conference on Software Engineerin (ICSE). pp. 222–231 (2013)
21. International, S.: SAE AS5506 Rev. C Architecture Analysis and Design Language (AADL). SAE International (2017)
22. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: 22nd ACM Symposium on Operating Systems Principles (SOSP). pp. 207–220 (2009)
23. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Spec. of Businesses and Systems, vol. 523, pp. 175–188. Springer (1999)
24. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 6355, pp. 348–370 (2010)
25. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert – a formally verified optimizing compiler. In: Embedded Real Time Software and Systems (ERTS) (2016)
26. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: 14th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 337–340 (2008)
27. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: 25th International Conference on Automated Deduction (CADE). LNCS, vol. 9195, pp. 378–388 (2015)
28. Naumann, D.A.: Observational purity and encapsulation. Theor. Comput. Sci. (TCS) **376**(3), 205–224 (2007)

29. Robby, Chalin, P.: Preliminary design of a unified JML representation and software infrastructure. In: 11th Formal Techniques for Java-like Programs (FTfJP). pp. 5:1–5:7 (2009)
30. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. In: 11th ACM SIGSOFT Symposium on Foundations of Software Engineering held jointly with 9th European Software Engineering Conference (ESEC/FSE). pp. 267–276 (2003)
31. Rodríguez, E., Dwyer, M.B., Flanagan, C., Hatcliff, J., Leavens, G.T., Robby: Extending JML for modular specification and verification of multi-threaded programs. In: 19th European Conference Object-Oriented Programming (ECOOP). LNCS, vol. 3586, pp. 551–576 (2005)
32. Thiagarajan, H., Hatcliff, J., Belt, J., Robby: Bakar Alir: Supporting developers in construction of information flow contracts in SPARK. In: 12th Source Code Analysis and Manipulation (SCAM). pp. 132–137 (2012)
33. Thiagarajan, H., Hatcliff, J., Robby: Awas: AADL information flow and error propagation analysis framework. Innovations in Systems and Software Engineering (ISSE) (2021), (to appear)
34. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. ACM Trans. Priv. Secur. (TOPS) **21**(3), 14:1–14:32 (2018)
35. Wilding, M., Greve, D.A., Richards, R.J., Hardin, D.S.: Formal verification of partition management for the AAMP7G microprocessor. In: Design and Verification of Microprocessor Systems for High-Assurance Apps., pp. 175–191. Springer (2010)
36. Yi, J., Robby, Deng, X., Roychoudhury, A.: Past expression: encapsulating pre-states at post-conditions by means of AOP. In: Aspect-Oriented Software Development (AOSD). pp. 133–144 (2013)
37. Zhang, Z., Robby, Hatcliff, J., Moy, Y., Courtieu, P.: Focused certification of an industrial compilation and static verification toolchain. In: 15th Software Engineering and Formal Methods (SEFM). LNCS, vol. 10469, pp. 17–34 (2017)
38. Project Loom. https://openjdk.java.net/projects/loom
39. Sireum Logika: A program verifier and a natural deduction proof checker for propositional, predicate, and programming logics. http://logika.v3.sireum.org
40. Slang: The Sireum Programming Language. http://slang.sireum.org