

Proof Engineering in Logika: Synergistically Integrating Automated and Semi-Automated Program Verification

Stefan Hallerstede, Robby, John Hatcliff, Jason Belt, and David Hardin

Aarhus University, Aarhus, Denmark
`sha@ece.au.dk`

Kansas State University, Manhattan, Kansas, USA
`{robby,hatcliff,belt}@ksu.edu`

Collins Aerospace, Cedar Rapids, Iowa, USA
`david.hardin@collins.com`

Abstract. Recent work on industry-capable program verification technology has emphasized the need for greater predictability in the performance of SMT-based automated verification approaches. Moreover, foundational limitations of SMT necessitate some incorporation of manual proof steps, and researchers are considering the utility of handing off some verification obligations to more powerful semi-automated interactive proof assistants. In this paper, we describe how capabilities that are usually associated with expert-level semi-automated proof assistants can be integrated synergistically in a developer-friendly code-based proof language to address many of the limitations of traditional SMT-based automated verification. Our approach enables proofs of more powerful properties to be carried out directly in a familiar programming environment rather than in separate proof assistant tools that often utilize low-level encodings of program semantics in annotations that are unfamiliar to industry developers. Because the proof language is implemented at the same level of abstraction as the programming language, using familiar syntax, our approach can provide easier-to-understand visualizations of rewriting/simplification steps that better align with the developer’s mental model of program execution (providing a better user experience). Our approach is implemented in the open-source Logika program verifier for Slang (a safety-critical subset of Scala). We evaluate the framework on a collection of examples, including libraries for high assurance embedded system data structures developed by engineers at Collins Aerospace.

1 Introduction

Program verification with SMT (Satisfiability Modulo Theories) solvers has made significant advances in recent years, enabling formal verification of increasingly complex software. However, there are several issues that frustrate developers and impede the adoption of formal verification techniques.

First, SMT-based program verifiers often suffer from “proof instability” (e.g. investigated by Zhou et al. [34]) – situations in which “semantically irrelevant changes to the query can have large effects on the SMT solver’s response”. E.g., simply renaming a variable might cause a previously verified procedure to take orders of magnitude longer to verify, or fail completely. Second, the effectiveness of SMT-based program verification depends heavily on solver heuristics and configuration choices to work around the general inherent incompleteness of SMT, creating significant usability challenges for developers [9]. Modern SMT solvers like Z3 and CVC5 employ hundreds of heuristic rules for, e.g.: (a) quantifier instantiation patterns, (b) theory combination strategies, (c) search-space prioritization, and (d) preprocessing transformations. These implementations differ substantially between solvers, leading to: (i) unpredictable performance profiles for similar verification tasks, (ii) non-portable verification results across solver versions (sometimes performance on certain tasks may regress [21]), and (iii) configuration sensitivity requiring expert-level tuning. To deal with the unpredictability as well as fundamental limitations of SMT, previous work has investigated handing off some verification obligations to more powerful interactive theorem provers, e.g., [27,23]. However, the community is struggling to find the most effective approaches for achieving these handoffs in ways that are easy to use, that achieve the best synergy between automated and semi-automated techniques, and that avoid disruptive steps in proof engineering workflows.

In this paper, we describe how capabilities (such as induction proofs for recursive data structures, term rewriting and simplification) that are usually associated with expert-level semi-automated proof assistants can be integrated in a developer-friendly code-level proof language. This stands in contrast to other “hand-off” approaches that require developers to complete the hand-off in a separate proof assistant tool that uses low-level encodings of program semantics in annotations that are unfamiliar to industry developers [19]. We show how these integrated capabilities can be synergistically combined with SMT to address the limitations of traditional SMT-based automated verification described above. Our approach is implemented in the open-source Logika program verifier for Slang. Previously, we gave an overview of Logika [32]. We described design goals for Logika, Logika’s contract language, the incorporation of Logika into the widely-used IntelliJ IDE to form an *integrated verification environment* (IVE), and gave a one-page summary of capabilities of the initial version of Logika’s proof language. In this paper, we report on our complete realization of the integrated proof language concept, and its application to industry-relevant examples. The specific contributions of this paper are as follows.

- We present our strategy for integrating associated semi-automated verification in an approach that seamlessly and synergistically integrates with Logika’s symbolic execution SMT-based verification.
- We describe the language’s support for inductive proofs for code that manipulates inductively defined data types.
- We illustrate explainability and visualization features that we have added in the IVE to help developer understanding of succeeding and failing proof steps.

- We describe Logika’s rewriting and simplification proof methods that use approaches similar to partial evaluation and symbolic execution to simplify programming language terms appearing in proofs. Because the proof language is implemented at the same level of abstraction as the programming language, using familiar syntax, our approach can provide easier-to-understand visualizations of rewriting/simplification steps that better align with the developer’s mental model of program execution.
 - We illustrate the features above using a collection of examples that includes an embedded system data structure library developed at Collins Aerospace.
- Slang and Logika are part of the Sireum framework for language processing, analysis, and verification developed at Kansas State University. The entire Sireum framework is publicly available [36] under an open source license, as are the code examples referenced in this paper [35].

2 Background

In this section, we summarize the rationale for Slang and Logika’s design, drawing from the more detailed presentations in [31,32,13].

Slang: The Slang (safety-critical) dialect of Scala was designed “hand in glove” with Logika to achieve effective, efficient, and usable verification while also supporting development for programming language and model processing, targeting embedded systems. Slang retains some of the expressive higher-level features of Scala (classes, traits, higher-order functions) while restricting them to a form that enables more effective verification. A subset of Slang (called “Slang Embedded”) is further restricted to constructs that can be translated to C and Rust appropriate for embedded systems without the need for dynamic memory allocation/automatic memory management. For a detailed overview of Slang features and design rationale, see [31].

Examples of Slang’s restrictions of higher-level Scala features include: (a) a modified type system that strictly separates immutable from mutable types, and (b) restrictions on mutable object aliasing, allowing aliasing to only be introduced in a single programming construct (i.e., method invocation) under certain object separation constraints. These customizations reduce developer reasoning effort and significantly simplify formal analyses (i.e., reducing verification costs). Slang’s *extension interfaces* (akin to those in the Bogor model checker [30]) allow Slang to interface with full Scala, Java or any other JVM-based language, and C/Rust libraries, as well as facilitating domain-specific customizations.

While Slang is a strict Scala subset, its programming language features are still rich enough to support large application development. The largest system implemented in Slang is Sireum itself (which includes Slang and Logika). This allows for self-application of Sireum tooling to its own Slang codebase. At this point, the Sireum codebase consists of 41 (Maven) modules with close to 391k lines of code as 84% Slang, 11.4% Scala, and 4.6% Java.¹

In addition to using Slang with standard JVM-based Scala/Java tools and ecosystems, the Slang Embedded subset can be transpiled to C without requiring

¹ <https://github.com/sireum/kekinian/tree/04afeba>

garbage collection at runtime (i.e., objects are globally/stack-allocated). For additional assurance, the translated C code can be compiled using the CompCert verified C compiler [26]. We are developing transpilation of Slang Embedded code and contracts to Rust (with Verus [24] contracts) as part of an ongoing DARPA PROVERS project led by Collins Aerospace.

Slang and Logika have primarily been used on industrial research projects on high-assurance model-based development at Collins Aerospace [6,7] and Galois [17,16]. Given an AADL [3] component-based system architecture model, the Sireum HAMR high assurance embedded system engineering framework generates AADL runtime services in Slang Embedded that can be deployed in various platforms, including the seL4 verified micro-kernel (via C) with formal evidence that architectural constraints are preserved, thus enabling guarantees of safe/secure inter-component spatial and temporal separations [6]. In collaboration with Collins Aerospace and seL4 developers, HAMR was used to build an experimental mission control subsystem running on seL4 for the Boeing CH-47 Chinook helicopter platform. Regarding the primary developer-facing tooling, the Sireum Integrated Verification Environment (IVE) – a customized version of IntelliJ IDEA – integrates various Sireum tools such as the Slang front-end (providing, e.g., type checking, refactoring, etc.), the Proyek incremental/parallel build tool, and Logika, all running as microservices in a background Sireum server. Moreover, we recently added VSCode integration for Sireum.

Logika: Slang’s contract language is based on classical logic and supports assertions, pre/post-conditions, data type invariants, and global invariants for global states. Verification of code conformance to contracts is performed compositionally and employs multiple back-end solvers in parallel, including Alt-Ergo [8], CVC4 [5], CVC5 [4], and Z3 [29]. In principle, other theorem provers could also be employed. To provide a continuous user abstraction experience, however, a developer should be supported within Slang/Logika as much as possible. Current work on Logika focuses on proof constructions within Slang to allow this. Logika uses a forward verification approach based on symbolic execution instead of a backward approach based on weakest pre-condition computation. Based on two decades of experience of implementing symbolic execution tools for both Java and SPARK, we believe that the symbolic execution approach produces diagnostic information about verification steps that is much easier to understand and also allows a more intuitive summary of the underlying verification algorithm for engineers and students. The scalability of Logika is complemented by using incremental, focused, and parallel (distributable) verification algorithms. Verification results, developer feedback on verification status, and contract/proof editing are supported in the Sireum IVE. With the IVE, we are able to support testing concepts with conventional unit libraries for Scala (e.g., ScalaTest) as well as provide testing coverage concepts using IntelliJ’s built in coverage facility.²

² See <https://doc.sireum.org/venues/presentations/logika/tccoe22/> for a video of a 25-minute technical talk and demonstration of Logika’s IVE user interface and server-based checking architecture.

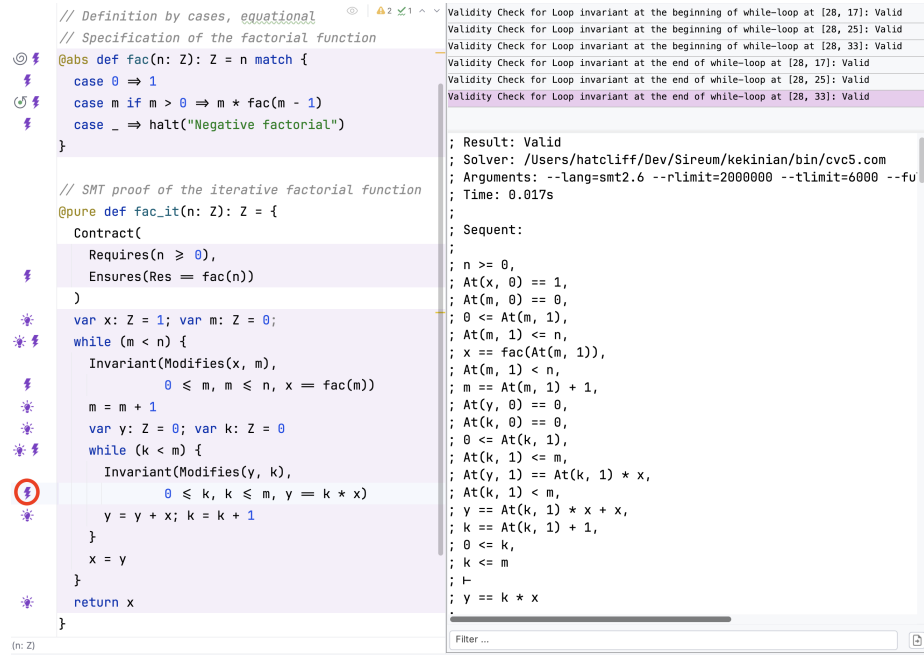




Fig. 1. Fully automated verification of factorial using Logika SMT-based symbolic execution

3 Proof Language Principles

In this section, we summarize basic features of Logika’s proof language using a very simple example – an iterative version of the factorial function implemented using addition as the only numerical operation. Figure 1 presents a version of the example in the Sireum IVE in which the proof language is not utilized, i.e., verification is performed automatically using SMT. At the top left of the function, a Slang (executable) specification function `fac` is defined. The Scala annotation `@abs` indicates that the function is “strictly pure” (it is a side-effect free construct that can be directly translated to the SMT-LIB expression language) and can be used in Logika specifications and proof contexts. However, in contrast to Slang’s similar `@strictpure` functions (see [31] for a detailed discussion of Slang’s function flavors), `@abs` functions will not be unfolded automatically unless explicitly requested in Logika’s rewrite tactics (similar to Isabelle’s `definition` and `fun` distinction as illustrated below). Then, the iterative version `fac_it` is defined with: (a) a contract precondition requiring the argument to be non-negative, and (b) a post-condition stating that the method return value (denoted as `Res`) is equal to the result of the specification function. As the developer types, Logika continuously runs in the background to check for possible run-time exceptions, assertion violations, and to verify that the code conforms to declared contracts.

Significant engineering efforts have been devoted to displaying verification results directly in terms of program features that the developer can recognize instead of lower level representations such as information flowing to/from the SMT

solver. The engineering involves mapping Logika’s three-address code intermediate representation and internal logic variables back to Slang-level program expressions and variables, and also maintaining mappings between program artifacts and SMT-LIB encodings. One of Logika’s most distinguishing features is to make this verification information available to developers at each program point in the code via clickable annotations in the left margin of the editor. There are two types of information: the lightbulb icons  display *facts* roughly corresponding to statement level pre/post-conditions; and lightning bolt icons  display sequents representing verification conditions that are encoded as calls to the underlying configured SMT solvers.

As Logika works, it collects facts that it discovers by symbolically moving forward step-by-step through the code. Some of the accumulated facts are immediately apparent from the structure of each program statement (we will refer to these as *immediate facts*). Others are the result of deductions that it has made by calling the underlying SMT solvers (we will refer to those as *deduced facts*). Logika can display all the inferred facts that it has accumulated at any point in the program (via the lightbulb icons) to provide valuable hints about how to reason systematically about the program. These Slang-level inferred facts are computed based on Logika’s internal symbolic execution path conditions that must hold at the particular program points, which are intuitive as they are directly in line with the regular (concrete) program execution.





The  annotations indicate the points at which Logika makes automated deductions that require interactions with its underlying SMT solvers. Logika terms these interactions as *summonings* because the power of SMT solving is being “summoned” to make a deduction that cannot be carried out using simple syntactic manipulation of the current facts. Clicking on  shows the details of the summoning. Generally, summonings occur for each line of a post-condition, for (implicit/explicit) assertions, invariants, for checking the pre-condition of a called method, and for code branches to determine the feasible path(s) along which verification should proceed. The right side of Figure 1 shows selected aspects of clicking on  (annotated with a red circle) at the second loop invariant. The loop invariant contains three (implicitly conjoined) clauses, and for each of these, Logika establishes two verification conditions (VCs, one that requires the clause to hold at the beginning of the loop, and another for the end of the loop). From the six total VCs for the selected , the user has selected the final one (the end of the loop VC, the clause $y == k * x$). The display provides information about the underlying SMT solver invocation (e.g., *cvc5* is invoked with the particular configured set of arguments). Below this, the sequent display is given with the relevant clause as the conclusion. The sequent antecedents hold the accumulated facts relevant to the verification. Annotations such as $\text{At}(k, 1)$ refer to the value of variable k as its second occurrence (zero-based counting) within the method.

Figure 2 presents verification excerpts of the same example with selected features of Logika’s manual proof language. The excerpts include the body of the inner **while** loop (an additional assignment to an intermediate variable *yn* has

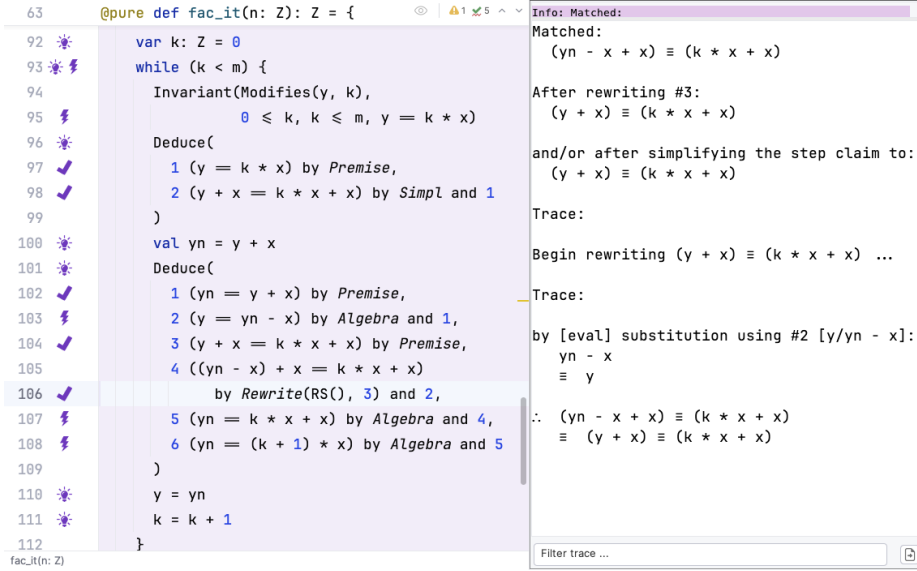






Fig. 2. Verification of factorial using proof language (excerpts)

been introduced to facilitate the illustration). Proof blocks are presented Logika’s `Deduce(. .)` construct, which includes a numbered list of proof steps. Each proof step has a claim and an associated justification (or tactic) to apply to prove the claim. In the first `Deduce`, the claim $y == k * x$ is justified by `Premise` because it follows immediately, since the invariant is assumed to be true at the top of loop body following the usual Hoare logic approach. In general, any claim shown in a  can be copy-pasted “as is” as `Premise` in the corresponding program point. In addition to serving as user feedback,  claims as `Premises` facilitate integration of Logika automation and interactive verifications. The second proof step is proved using `Simpl`, which rewrites terms using known facts in the form of equations and capturing the semantics of Slang expressions (akin to partial evaluation). Using the operator `and`, the justification `Simpl` can be restricted to equality claims supplied as arguments (the claim of proof step 1 in this case). Note that the claim is proven without an external call to the SMT solvers (indicated by having the “check mark”  icon instead of ) , thus illustrating one of the avenues for proof engineers to avoid the potential instability and increased overhead of SMT. `Simpl` can also be used without any `and` argument, in which case it can use any preceding proof step claim in scope for equality substitutions. We recently introduced `ESimpl` (not shown), which offers a backtracking variant of `Simpl` that can recover from detrimental substitutions that diverge from proving the stated proof step claim. Table 1 lists some of Logika’s justifications.

In the second `Deduce`, the second claim is proved using `Algebra`. This step is proven by an external call to SMT (as indicated by the bolt gutter annotation), but it uses SMT in a very limited way that is less costly and likely more stable: it only uses explicitly referenced claims (the claim from step 1 in this case) and does not include implicitly any facts from the context. Proof of the fourth claim

Justification	Informal description
Simpl	Simplify current clause in proof into known fact, or fail
ESimpl	Backtracking version of the above
Rewrite	Rewrite indicated clause in proof into current clause, or fail
Premise	Known fact
Subst	Substitute one term for another left-to-right “>” or right-to-left “<”
AllI, AllE	Complete set of natural deduction rules (not all shown here)
Auto	Summon unconstrained configured SMT solvers
Algebra	Summon SMT solvers specialized to unquantified arithmetics and logics

Table 1. Logika manual proof step justifications

illustrates Logika’s **Rewrite** justification. In general, **Rewrite** takes an ordered rewrite set consisting of lemmas/theorems and references to **@abs** methods to unfold (not shown), and a proven claim to be rewritten. For the fourth claim, rewriting is applied to claim 3 and the rewrite set **RS()** is empty, but the rewriting is specified to utilize the equality claim 2 as a left-to-right rewrite rule. Rewrite sets **RS** are defined using Slang sequence types and can be specified inline or by given an annotated Slang value definition **@rw val**. Set union and difference work on **RS**, e.g., **myRewriteSet ++ otherSet - RS(m _)**, which can also be stored in a **@rw val** or inlined. In practice, **RS** members are names of Slang definitions.

Thus, **Rewrite** gives yet another way to avoid invoking SMT for a proof. Logika’s rewriting approach is similar to Isabelle’s, but it is currently not as powerful. However, it does have several of the key capabilities (flexible definition of rewrite sets with controlled unfolding of definitions). Even in its current form, we have found that it is powerful enough to address most common scenarios. By avoiding having to export VCs using hard-to-understand encodings to an external theorem prover, the developer works directly in terms of the programming language syntax, definition constructs, and abstraction level in the same programming/verification environment. In addition, clicking on ✓ annotation for the claim provides a trace of the rewriting directly in terms of the program-level constructs. The right side of Figure 2 shows the rewriting trace for claim 4. First, the trace illustrates that claim 3 is to be rewritten. Then, it is written using the indicated equality of claim 2 (left to right), by substituting **y** for **yn - x**. This yields a match with claim 4, thus proving the claim.

4 Case Study

To assess the utility of Logika’s proof language, we used it to prove the correctness of several data structure libraries. One of those is a Slang implementation of a doubly-linked linked list (DLL) based on similar approaches used in embedded security devices at Collins Aerospace [14]. The initial Slang implementation was completed several years ago with only a few simple aspects verified because it was difficult to complete the verification with the earlier implementations of Logika. The recent Logika addition of proof language features and better support for abstraction and refinement has enabled us to significantly expand the scope of the proofs.

Abstractly, a list can be easily expressed algebraically using the constructors **Nil** and **Cons** for values of type **List**. However, for high assurance embedded devices, it is undesirable to use such list directly because its manipulation requires dynamic memory allocation, which introduces unpredictable behaviors.


```

1 // DLL Node type
2 @datatype class Node[E](elem: E, used: B, left: DLLPool.PoolPtr, right: DLLPool.PoolPtr) {}
3
4 // DLL Pool
5 object DLLPool {
6   type PoolPtr = Z
7   type PoolMem[E] = MSZ[Node[E]]
8   val Null: PoolPtr = -1
9   //...(excerpts of helper properties for concrete representation)
10  @abs def isPointer[E](pool: PoolMem[E], p: PoolPtr): B = { p == Null || pool.isInBound(p) }
11  @abs def isValidPointer[E](pool: PoolMem[E], p: PoolPtr): B = { pool.isInBound(p) }
12  @abs def freeNodesProp[E](pool: PoolMem[E], free: Z): B = { free == count_free(pool) }
13  //...(excerpts of properties used in abstraction/refinement relation)
14  @abs def asList[E](pool: PoolMem[E], head: PoolPtr): List[E] =
15    if (isValidPointer(pool, head)) {
16      Cons(pool(head).elem, asList(pool, pool(head).right))
17    } else { Nil() }
18  //...
19 }
20
21 @record class DLLPool[@imm E](eDefault: E, poolSz: Z) {
22   val defaultNode: Node[E] = Node[E](eDefault, F, Null, Null)
23   val maxSz: Z = if (poolSz > 0) poolSz else 0
24   val pool: PoolMem[E] = MSZ.create(maxSz, defaultNode) // pool storage
25   var free: Z = maxSz // current # of free items in pool
26   var head: PoolPtr = Null // index of the current logical head Node
27   var tail: PoolPtr = Null // index of the current logical tail Node
28   // ...(excerpts of invariants for concrete representation)
29   @spec def freeNodes = Invariant( freeNodesProp(pool, free) )
30   // Type & function definitions (with contracts) for abstract list (see Figure 6 for example)
31   // ...
32   // Specification of abstraction relations (omitted)
33   // ...
34   // Operations (with contracts) for concrete representation (omitted)
35   // ...
36 }

```

Fig. 3. DLL excerpts - primary declarations and concept outline

Instead, we use such list and its accompanying theorems indirectly as abstract specification and in proof only. The data representation chosen for the concrete DLL implementation is challenging for understanding and verification in that it is not obvious that a list is implemented. The logical ordering of elements in the DLL does not follow their physical ordering in pool memory. Allocating new element storage and subsequent reclaiming must be managed explicitly. There are also several fairly complex invariants that need to be maintained by each DLL operation.

4.1 Overview

Figure 3 presents some of the primary definitions in the DLL library. In the concrete DLL implementation, in order to avoid dynamic memory allocation and guarantee locality of all memory access, all data of the DLL is stored in a memory block of fixed size represented as a Slang mutable sequence (line 24). The pool holds items of type `Node`. Each node (line 2) includes a data element `elem` of type `E`, indices of Nodes to the logical `left` (towards the head of the list) and `right` (towards the tail of the list). `head` and `tail` are pool indices for the current logical head and tail of the list (see Fig. 4).

Our verification strategy is as follows. Approximately 13 invariants are specified on the DLL structure. For example, Figure 3 line 29 illustrates a simple

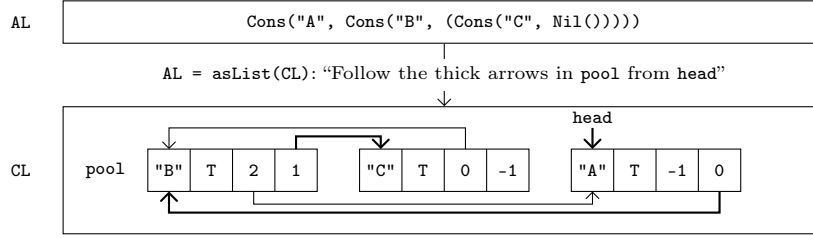


Fig. 4. DLL data structure and its list abstraction

invariant **freeNodes** that specifies that the pool variable **free** that holds a count of the number of free nodes in the pool matches the count computed by traversing the memory blocking and counting the nodes marked unused (note that the invariant uses the **freeNodesProp** helper property defined earlier in the figure). All DLL operations are proved to maintain the invariants. The **DLLPool** object contains approximately 20 helper methods with specification and proofs (starting at line 10, Figure 3 shows some of the simpler ones like **isPointer**).

Next, we create an abstract operational specification of the DLL using an easily expressible and understandable inductively defined **List** type (for lack of space, these definitions are omitted since they correspond to the familiar cons-list data structure). Operations on **List** are written using the pure functional programming language features of Slang. This functional modeling style is common in theorem proving based on type theory.

Subsequently, a family of relations **asList** are defined as Slang specification methods that relate concrete DLL values to abstract **List** values (one of the methods is shown in Figure 3 line 14). Figure 4 illustrates the relationship between the concrete and abstract values. At the top of the figure the abstract list is shown as an expression, at the bottom the concrete DLL implementation is shown where the **left** and **right** fields of a node point to other locations in the mutable sequence **pool**. The refinement relation between the two representations consists of reconstructing the abstract list by following the **right** pointers from the **head** onwards. Subsequently, the concrete implementation of the DLL is verified against the abstract list model using refinement. In other words, we show that the concrete implementation is a simulation of the abstract list.

Figure 5 illustrates key concepts of the proof approach. Abstract list **AL** and concrete (implementation) list **CL** stored in the buffer are related by the relationship **AL = asList(CL)**, describing the simulation of **AL** by **CL**. This simulation is used to relate abstract functions such as **length** that yields the length of a list to the **CL** function **sizeOf** that yields the length of a **CL**. In order for **sizeOf** to implement **length** correctly, the value returned by **CL.sizeOf** for a list must be equal to that **AL.length** if **AL = asList(CL)**. (This is addressed in Figure 8, discussed later). Similarly, the **CL.cons** simulates the abstract constructor **Cons** of the abstract list if the buffer is not full. (This is addressed in Figure 9, discussed later).

Generally speaking, proofs in the library dealing with more abstract concepts are more algebraic relying on rewriting, while pure implementation-level proofs require more complex properties that cannot be cast in the form of equations.

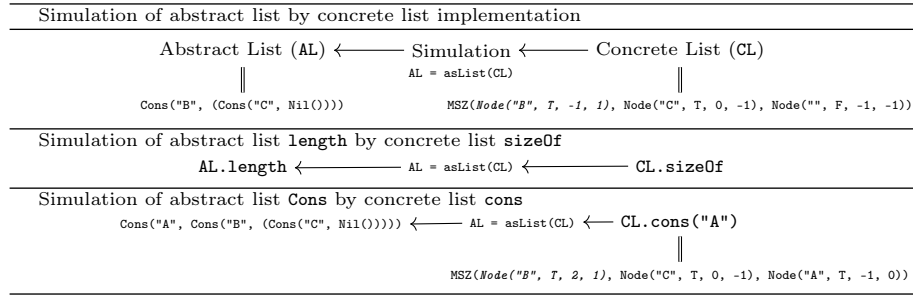


Fig. 5. Case Study Proof Approach

The refinement relationship is an invariant of the concrete implementation and by mentioning both abstraction levels requires a mix of algebraic proofs and proofs that use properties with more complex shapes. In our experience, SMT solvers require some help from the user in order to achieve such proofs. In the approach that we follow, Slang's flexibility enables proofs to be carried out in supporting objects as much as possible and not in the code so as not to blur the implementation with more complex proofs.

4.2 Programming and Proof

The development and proof of the abstract list with its concrete implementation as doubly-linked list is separated into three parts:

- (1) The abstract list modeled as an algebraic datatype `List[E]` with constructors `Nil()` and `Cons(value, list)`, where `value` of generic type `E` and `list` of type `List[E]`. Properties of abstract lists are proved in corresponding `List` companion object (see theorem `length_impl_with_acc_sum` in Figure 6).
- (2) A doubly-linked list companion object `DLLPool` (see Figure 3) with theorems relating to implementation concerns, the theorems about refinement, and related executable specification functions.
- (3) The doubly-linked list class `DLLPool[E]` (see Figure 3) with the implementation code and correctness proofs expressed in terms of theorems proved in (1) and (2).

Abstract list properties. Properties of abstractions are often needed to support refinement proofs. For instance, the refinement proof of `CL.sizeOf` (see Figure 8) needs to establish that the result `Res` equals `AL.list.length`, the length of the corresponding abstract list. In the `CL` operation, `Res` is iteratively computed by a loop using a local variable `res`. To achieve `res == list.length` on termination, the progress in the computation of `res` must be recorded. This can be stated as the universal claim $\text{All}\{(\text{acc}: \mathbb{Z}) \Rightarrow \text{acc} + \text{l.length} == \text{l.len_acc}(\text{acc})\}$. To help with proving the claim, we introduced the `length_impl_with_acc_sum` Logika method theorem shown in Figure 6. The theorem is proven by using Logika's `@induction` (line 3). In the base case of `Nil()`, the theorem claim can be discharged automatically by SMT solving, thus there are no proof annotations (line 20). The inductive step of `Cons(...)` (line 4) illustrates the use of Logika's simplification and natural deduction tactics. The previous declaration

```

1 @pure def length_impl_with_acc_sum[T](l: List[T]): Unit = {
2   Contract(Ensures(All{(acc: Z, bcc: Z) => bcc+1.len_acc(acc) == 1.len_acc(bcc+acc)}))
3   (l: @induct) match {
4     case Cons(v, n) =>
5       Deduce(
6         1 (All{(acc: Z, bcc: Z) => bcc+n.len_acc(acc) == n.len_acc(bcc+acc)}) by Premise,
7         2 (Cons(v, n).length == 1 + n.length) by Simpl,
8         3 Let((acc: Z) => SubProof(
9           4 Let((bcc: Z) => SubProof(
10            5 (Cons(v, n).len_acc(bcc + acc) == n.len_acc(1 + (bcc + acc))) by Simpl,
11            6 (All{(bcc: Z) => bcc+n.len_acc(1+acc)==n.len_acc(bcc+1+acc)}) by AllE[Z](1),
12            7 (bcc + n.len_acc(1 + acc) == n.len_acc(bcc + (1 + acc))) by AllE[Z](6),
13            8 (bcc + Cons(v, n).len_acc(acc) == bcc + n.len_acc(1 + acc)) by Simpl,
14            9 (bcc+Cons(v, n).len_acc(acc) == Cons(v, n).len_acc(bcc+acc)) by Auto,
15            10 (bcc + 1.len_acc(acc) == 1.len_acc(bcc + acc)) by Auto
16          )),
17          11 (All{(bcc: Z) => bcc+1.len_acc(acc) == 1.len_acc(bcc+acc)}) by AllI[Z](4)
18        )),
19        12 (All{(acc: Z, bcc: Z) => bcc+1.len_acc(acc) == 1.len_acc(bcc+acc)}) by AllI[Z](3))
20     case Nil() => return
21   }

```

Fig. 6. Example AL operation (high-level specification for DLL)

of the `@induct` proof strategy (line 3) causes the induction hypothesis to be implicitly introduced in the fact set when in the `Cons` proof case (in the IVE, the fact would show up in \star annotation information). Because of this, the explicit statement of the induction hypothesis at line 6 can be discharged using the `Premise` justification. Overall, some of the proof steps appear obvious, but none of the simplification steps using `Simpl` can be discharged by SMT solving due in part to the presence of uninterpreted functions, quantifier manipulation, etc. in the Slang-to-SMT encoding. To help direct SMT solving, we manually eliminate/instantiate and re-introduce the universal claim by using Logika universal quantifier elimination and introduction (`AllE`/`AllI`) rules (see lines 11 and 17). With quantifiers removed, and by judicious use of `Simpl` to prove facts to help SMT along, other parts of proof can be discharged via SMT solving using the `Auto` justification (lines 14 and 15).

This example illustrates nicely the common interplay between Logika SMT-based automated verification and interactive theorem proving found in many other parts of the DLL code, where automation is used as best as can be afforded using SMT solvers, coupled with some interactive proofs done using Logika’s rewriting system. One very attractive aspect of our approach is that these activities are all integrated in a single tool (the IVE – built on a widely used IDE) and carried out using program-level features (instead of lower-level language encodings as they would appear in a separate theorem proving tool).

Implementation and refinement properties. Properties concerning the implementation of the doubly-linked list are stated and proved in the companion object `DLLPool` declared in Figure 3 line 5. Figure 7 gives one example of such a property `refines_p_not_Nil` that states that if a provide concrete representation satisfies the refinement relation and there is a valid pointer `p` (i.e., the pointer does not reference an unused node), then the abstract list is not equal to `Nil`. Intuitively, the presence of rewriting and simplification tactics in the proof indicates that

```

1 @pure def refines_p_not_Nil[E](pool: PoolMem[E], p: PoolPtr, l: List[E]): Unit = {
2   Contract(Requires(isValidPointer(pool, p), refinesProp(pool, p, l)), Ensures(l != Nil[E]()))
3   Deduce(
4     1 (isValidPointer(pool, p)) by Premise,
5     2 (pool.isInBound(p)) by Rewrite(RS(isValidPointer _), 1),
6     3 (refinesProp(pool, p, l)) by Premise,
7     4 (asList(pool, p) == l) by Auto and 3,
8     5 (asList(pool, p) == Cons[E](pool(p).elem, asList(pool, pool(p).right)))
9     by RSimpl(RS(asList _)) and (1, 2),
10    6 (Cons[E](pool(p).elem, asList(pool, pool(p).right)) == l) by Subst_<(5, 4)) }

```

Fig. 7. Example low-level theorem from DLL companion object

```

1 @pure def sizeOf: Z = {
2   Contract(Requires(refinesProp(pool, head, list)),
3     Ensures(refinesProp(pool, head, list), Res == list.length))
4   var res: Z = 0
5   var p = head
6   @spec var l = list
7   Spec { length_impl_with_acc(list) } // apply lemma as method invocation
8   while (!isLeaf(p)) {
9     Invariant(Modifies(res, p, l),
10      refinesProp(pool, p, l), isPointer(pool, p), l.len_acc(res) == list.length)
11     Spec { length_impl_with_acc(l.tl); refines_p_not_Nil(pool, p, l)
12      refines_p_sublist(pool, p, l) }
13     res = res + 1
14     p = pool(p).right
15     Spec { l = l.tl }
16   }
17   return res
18 }

```

Fig. 8. Example low-level side-effect free CL method (with refinement proof)

the claim to be proved is easily deduced by appealing to involved equalities, e.g., from function definition `isValidPointer` in line 5 in Figure 7. The equality is a consequence of the definition of `isValidPointer` in line 11 in Figure 3. Property `refines_p_not_Nil` is used in the refinement proof of function `sizeOf` below that needs to interpret low-level properties like `isValidPointer(pool, p)` in terms of high-level properties like `l != Nil[E]()`.

Implementation of side-effect-free CL methods. We next illustrate in Figure 8 the proof of a CL method `sizeOf` that traverses but does not modify the CL structure. Although the method uses imperative features, it has no visible external side effects and so is annotated with the Slang `@pure` annotation. The postcondition indicates that, given that `list` is the abstract representation of the CL instance, `sizeOf` implements the abstract function `list.length`. The method contract also indicates that the refinement condition `refinesProp(pool, head, list)` is preserved. In the method body, the incrementally computed size of the DLL is stored in variable `res`. In order to relate the current value of `res` to the sublist it corresponds to, a `@spec` variable `l` is declared (line 6) that remembers the part of the corresponding abstract list that has not yet been visited (by leveraging Slang’s copy-on-write semantics on mutable objects). The entire proof has been factored out into a set of lemmas. The lemma applications are stated as developer-friendly method invocations that are placed inside `Spec` blocks. Such lemma applications can be confirmed by inspecting the \star claims after the application program points (not shown). The final `Spec` block in line 15 only

```

1  def cons(elem: E): Unit = {
2    Contract(Modifies(list),
3      Case(Requires(free > 0, refinesProp(pool, head, list)),
4        Ensures(refinesProp(pool, head, list), list == Cons(elem, In(list)))),
5      Case(Requires(free <= 0, refinesProp(pool, head, list)),
6        Ensures(refinesProp(pool, head, list))))
7    if (free > 0) {
8      if (isEmpty) {
9        head = 0; tail = 0
10       @spec val qool = pool
11       pool(0) = Node(elem, T, Null, Null)
12       Spec { count_free_on_alloc(qool, pool, 0); list = List.make(elem) }
13     } else {
14       val pnew: Z = findFreeNode()
15       @spec val qool = pool
16       pool(pnew) = Node(elem, T, Null, head)
17       Spec { unused_inv(qool, head, list, pool, pnew)
18         refines_new_head(pool, head, list, pnew, elem)
19         count_free_on_alloc(qool, pool, pnew) }
20       @spec val rool = pool
21       pool(head) = pool(head)(left = pnew)
22       Spec { list_coincidence(pool, rool, head); free_coincidence(pool, rool) }
23       head = pnew
24       Spec { list = Cons(elem, list) }
25     }
26     free = free - 1
27   }
28 }

```

Fig. 9. Example low-level side-effecting CL method (with refinement proof)

contains the update of the abstract list $l = l.tl$ that corresponds to the low-level assignment $p = pool(p).right$ in line 14, maintaining the relationship $refinesProp(pool, p, l)$ stated in the invariant (see line 10). The use of the lemmas keeps the “noise” in the programs code produced by proof low. It also makes the interaction with the SMT solvers more stable due to the smaller amount of claims involving only the contracts of the involved method theorems but not the contained proofs, which reduces the load of the solvers.

Implementation of side-effecting CL methods. Figure 9 shows the low-level CL function `cons` of DLL, which corresponds to the abstract list constructor `Cons`. Because the pool memory is limited in size, a call of `cons` only extends the CL if there is still space. This is expressed by the two cases of the contract of `cons`: the case `free > 0` permits extending the list, the case `free <= 0` leaves the CL unchanged. Note that `free` is proved to be the number of unused cells of the pool memory by means of the invariant `freeNodes` (see Figure 3, line 29). The condition is reflected by the leading if-statement in line 7.

The library includes a number of other methods for finding items, and inserting and deleting items, and the same proof strategies are applied for those.

5 Illustrations

In addition to the full artifacts for the DLL example presented in this paper, we have prepared examples of varying complexity to further illustrate Logika’s proof language features (see [35]).

- *Abstract list*: Inductive proof of $l = l.tl.tl$ for any list l . This development shows how a common inductive proof is carried out in Logika relying on rewriting and simplification to guide the proof.
- *Sequence sum*: In Logika, sequence induction is often done using while loops. This development shows how recursive properties are used in such proofs and how abstract properties are propagated to refinements.
- *Maximum of sorted sequence*: The maximum of a sequence of increasing values can be computed by returning the last value of the sequence. The proof of this is a program that computes the maximum value, confirming that the last value is the maximum. Except for the returned maximum, the entire program is enclosed in a `Spec` block. The program itself becomes a correctness annotation similar to typing information that confirms that values have the correct type.
- *Symbol table*: This example provides a symbol table that one might use, e.g., in a program/model implementation environment. The development demonstrates the use of function calls as theorem references and the replacement of abstract predicates in pre-conditions by more efficient implementations without affecting the difficulty of the proofs. This approach is useful when compile-time and run-time-verification use are combined in practice.

Due to space constraints, in this paper we have focused on illustrating concepts. Full evaluations of efficiency gains and usability are part of our plans for future work. It is worth emphasizing, however, that we believe Logika provides a uniquely high degree of usability by the fact that it provides a continuous of abstraction on its streamlined automated and interactive theorem proving approach at the level that is familiar to regular system engineers in realistic development workflows and supported by industrial-scale programming/verification environments. Regarding efficiency, one of the motivations for our work was to provide pathways for reducing the time and instability of verification. Anecdotally, besides failing to discharge VCs in numerous situations, our early SMT-heavy versions of the case study code required minutes for full verification. By using simplification/rewriting instead of SMT calls and by using the proof language to minimize the size of constraint sets sent to SMT, we were able to not only succeed in verification, but also to reduce verification time of this example to a few seconds.

6 Related Work

The Why3 framework [10] is a good example of previous work that aims to provide support for both SMT and interactive theorem proving (ITP). It provides an intermediate language (WhyML - combining both imperative and functional features) for encoding behavior and specifications, and a VC generator that generates VCs dischargable using automated calls to a variety of SMT solvers (Z3, CVC4/CVC5, and Alt-Ego) or exportable to the Coq interactive theorem prover. Coq proof scripts for Coq-proven VCs can be re-incorporated back into the WhyML artifacts. SPARK 2014 [20] (a contract-based specification and verification framework for a safety-critical subset of Ada) is an example of a powerful industrial verification framework that uses Why3 as its verifica-

tion engine. It translates SPARK programs into Why3 and relies on Why3 VC generation and verification framework to prove that SPARK programs conform to program contracts. This architecture provides combined automated SMT and theorem proving verification for SPARK programs. The difference with our work is that we aim to integrate SMT and *targeted* ITP directly within the programming language. Why3 is designed as an intermediate language and does not have the full support for developing, debugging, and execution that Slang does. When ITP is used, one must work with encodings of Why3 in Coq, whereas in Slang the developer works directly at the programming language level. When using SPARK 2014, diagnostic information is expressed in terms of Why3 encodings instead of directly in terms of the programming language. Moreover, when interactions with ITP are needed, users must understand a double-encoding – the encoding from SPARK 2014 to Why3, and then Why3 to Coq. An advantage of the SPARK 2014/Why3 approach is that one has access to the full power of a relatively mature theorem prover, whereas for our approach, we are currently providing a targeted set of tactics. As noted in [32], Logika’s extension architecture can facilitate exporting VCs to any theorem prover in future work.

Dafny [25], Frama-C [22], AutoProof [11] and Verus [24] are examples of program verification frameworks used in industry that also aim to incorporate some notion of a proof language phrased in programming idioms. While these tools have many attractive features, including some that are not supported by Logika, their proof languages are not nearly as expansive as what we provide in Logika. AutoProof integrates verification support to Eiffel tapping into its contract language. Both Dafny and Verus provide specification lemmas (with no executable code) whose pre/post-conditions can be proved using SMT and then reused in other program contexts. Dafny supports calculation blocks that can include “hints” to lower-level SMT solvers. Verus authors are developing an approach to translate VCs to the Lean ITP, which the developer needs to discharge interactively in Lean [28]. This has the same tradeoffs (dealing with encodings and working in a separate less-user-friendly tool vs. full ITP power) as with the SPARK / Why3 architecture described above compared to our approach. Frama-C provides verification for C programs. Via a plug-in mechanism powerful verification techniques like abstract interpretation are available. Contract and verification annotations are kept in the comments. This makes a tight integration of programming and proof concepts difficult. SED [18] uses annotations in comments and provides graphical visualizations of verification artifacts.

With a different objective than Logika, some of the approach to proof taken here has been used with Event-B [1] in the Rodin tool [2]. One of the design objectives of Rodin compared to its predecessor was to decompose large proof obligations and structure the remaining large proofs in such a way that the remaining sequents could be easily discharged by automated provers. Although Rodin is a tool for abstract modeling, the same approach to proof is at work in the presented work on Logika. It is not intended to eliminate the use of SMT (and other automated provers) from Logika but to make their use much more reliable and predictable.

7 Conclusion

Despite requiring “manual effort” and some level of expertise in formal methods, we believe there are significant benefits in integrating semi-automated proofs with automated program verifiers. A primary benefit is providing a pathway to continue verification progress when SMT-based automation becomes unstable or simply cannot prove true claims. Our goal in this paper has been to describe one particular approach that incorporates a proof language directly in the programming language and illustrate that it can work synergistically with SMT verification. We believe our approach is promising because the proof language emphasizes: (a) developer-friendly syntax and programming-like idioms, and (b) tool feedback like simplification/rewrite traces and SMT deductions are expressed directly in terms of program features (not lower-level encodings). We believe this approach is novel and applicable to other SMT-based automated program verifiers.

Regarding usability, an interesting anecdotal observation is that we teach Logika’s full proof language as illustrated here in master’s level courses in Aarhus University [13]. Logika’s basic natural deduction steps and substitution are taught in an undergraduate programming logic course at Kansas State University, which has included 1000 students during the last six years (see [33] for online textbook). While we would not necessarily expect all industry engineers that apply Logika or other program verifiers to use the proof language features, one can imagine that having an integrated proof language makes it easier to hand-off to verification engineers that are capable of applying the features. Moreover, being able to have engineers working in a single environment (instead of having to hand off to an external theorem prover that has completely different abstractions/notations) makes user workflows smoother and artifact management easier.

There are likely other theorem proving techniques or finer-grain controls over deduction that could be added to our framework. We have focused on rewriting and simplification; combinations of tableau-based proof search and resolution-like techniques as in Isabelle’s **blast** and **fastforce** tactics, which may be helpful for dealing with quantifier manipulation (which SMT-based solvers often struggle with), might be useful for adoption in Logika in the future. In general, Logika’s extension architecture allows one to realize custom proof tactics as needed.

We are continuing applications on other industry-related examples. This includes verifying the correctness of the HAMR [15] run-time libraries providing real-time threading and communication being used in the Collins Aerospace DARPA PROVERS INSPECTA project, by using a similar refinement proof strategy presented in this paper. That is, we are proving that the Slang-based implementation of the libraries (deployed to, e.g., seL4 via C/Rust) are a refinement of a Slang purely functional executable reference semantics for the subset of AADL supported by HAMR that have also been formalized in Isabelle [12].

Acknowledgments. This work was primarily funded by a DARPA SBIR Phase 2 SIRFUR award, with some support from the DARPA CASE and PROVERS projects.

References

1. Abrial, J.R.: *Modeling in Event-B – System and Software Engineering*. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.* **12**(6), 447–466 (2010)
3. of Automotive Engineers, S.: *Architecture analysis & design language (AADL)*. Aerospace Standard AS5506 (2004)
4. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength SMT solver. In: *TACAS 2022*. pp. 415–442. Springer (2022)
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: *International Conference on Computer Aided Verification (CAV)*. pp. 171–177. Springer (2011)
6. Belt, J., Hatcliff, J., Robby, Shackleton, J., Carciofini, J., Carpenter, T., Mercer, E., Amundson, I., Babar, J., Cofer, D., Hardin, D., Hoech, K., Slind, K., Kuz, I., Mcleod, K.: Model-driven development for the seL4 microkernel using the HAMR framework. *Journal of Systems Architecture* (2022)
7. Cofer, D.D., Amundson, I., Babar, J., Hardin, D.S., Slind, K., Alexander, P., Hatcliff, J., Robby, Klein, G., Lewis, C., Mercer, E., Shackleton, J.: Cyberassured systems engineering at scale. *IEEE Secur. Priv.* **20**(3), 52–64 (2022)
8. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-ergo 2.2. In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories* (2018)
9. De Moura, L., Bjørner, N.: The strategy challenge in smt solving. In: *Proc. of the Int. Conference on Automated Deduction (CADE)*. pp. 15–44. Springer (2011)
10. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: *Programming Languages and Systems. LNCS*, vol. 7792, pp. 1–16. Springer (2013)
11. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: Autoproof: auto-active functional verification of object-oriented programs. *International Journal on Software Tools for Technology Transfer* **19**(6), 697–716 (2017)
12. Hallerstede, S., Hatcliff, J.: A mechanized semantics for component-based systems in the HAMR AADL runtime. In: *19th International Conference on Formal Aspects of Component Software (FACS)*. LNCS, vol. 14485, pp. 45–64. Springer (2023)
13. Hallerstede, S., Hatcliff, J., Robby: Teaching with logika: Conceiving and constructing correct software. In: *FMTea 2024*. p. 106–123. Springer (2024)
14. Hardin, D., Slind, K.: Using ACL2 in the design of efficient, verifiable data structures for high-assurance systems. In: *ACL2 Theorem Prover and its Applications (ACL2-2018)*. EPTCS, vol. 280, pp. 61–76 (2018)
15. Hatcliff, J., Belt, J., Robby, Carpenter, T.: HAMR: An AADL multi-platform code generation toolset. In: *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. LNCS, vol. 13036, pp. 274–295 (2021)
16. Hatcliff, J., Belt, J., Robby, Legg, J., Stewart, D., Carpenter, T.: Automated property-based testing from aadl component contracts. In: Cimatti, A., Titolo, L. (eds.) *Formal Methods for Industrial Critical Systems* (2023)
17. Hatcliff, J., Stewart, D., Belt, J., Robby, Schwerdfeger, A.: An AADL contract language supporting integrated model- and code-level verification. In: *Proceedings of the 2022 ACM Workshop on High Integrity Language Technology. HILT '22* (2022)
18. Hentschel, M., Bubel, R., Hähnle, R.: The symbolic execution debugger (sed): a platform for interactive symbolic execution, debugging, verification and more.

- International Journal on Software Tools for Technology Transfer **21**(5), 485–513 (2019)
19. Ho, S., Protzenko, J.: Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* **6**(ICFP), 711–741 (2022)
 20. Hoang, D., Moy, Y., Wallenburg, A., Chapman, R.: SPARK 2014 and GNATprove. *International Journal on Software Tools for Technology Transfer* **17**(6) (2015)
 21. Jackson, D., Nelson, T., Schmitz, P.: Zelkova: SMT-Based policy analysis at scale. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. pp. 1–15. ACM (2020)
 22. Kosmatov, N., Prevosto, V., Signoles, J. (eds.): *Guide to Software Verification with Frama-C – Core Components, Usages, and Applications*. Springer (2024)
 23. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: *Witchel, E., Rossbach, C.J., Arpaci-Dusseau, A.C., Keeton, K. (eds.) Proceedings of the ACM SIGOPS 30th Symp. on Operating Systems Principles*. pp. 438–454. ACM (2024)
 24. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying Rust programs using linear ghost types. *Proc. ACM Program. Lang.* **7**(OOPSLA1), 286–315 (2023)
 25. Leino, K.R.M.: *Program Proofs*. The MIT Press (2023)
 26. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert—a formally verified optimizing compiler. In: *ERTS 2016* (2016)
 27. Martínez, G., Ahman, D., Dumitrescu, V., Giannarakis, N., Hawblitzel, C., Hriṭcu, C., Narasimhamurthy, M., Paraskevopoulou, Z., Pit-Claudel, C., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N.: Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In: *Caires, L. (ed.) Programming Languages and Systems*. pp. 30–59. Springer (2019)
 28. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: *CADE 28. LNCS*, vol. 12699, pp. 625–635. Springer (2021)
 29. Moura, L.d., Bjørner, N.: Z3: An efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. pp. 337–340. Springer (2008)
 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. ACM (2003)
 31. Robby, Hatcliff, J.: Slang: The Sireum Programming Language. In: *ISoLA 2021*. pp. 253–273. Springer (2021)
 32. Robby, Hatcliff, J., Belt, J.: Logika: The Sireum Verification Framework. In: *Haxthausen, A.E., Serwe, W. (eds.) FMICS*. pp. 97–116. Springer (2024)
 33. Thorton, J.: *Logical Foundations of Programming* (online textbook for KSU CS 301), <https://textbooks.cs.ksu.edu/cis301/index.html>
 34. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT Instability in Automated Program Verification. In: *Nadel, A., Rozier, K.Y. (eds.) FMCAD 2023*. pp. 178–188. TU Wien Academic Press (2023)
 35. Logika proof language case studies repository. <https://github.com/santoslab/logika-proof-language-case-studies>
 36. Sireum website, <https://sireum.org/>