A Language for Building Verified Software Components

Gregory Kulczycki, Murali Sitaraman, Joan Krone, Joseph E. Hollingsworth, William F. Ogden, Bruce W. Weide, Paolo Bucci, Charles T. Cook, Svetlana Drachova, Blair Durkee, Heather Harton, Wayne Heym, Dustin Hoffman, Hampton Smith, Yu-Shan Sun, Aditi Tagore, Nighat Yasmin, and Diego Zaccai

Technical Report RSRG-13-01
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

February 2013

Copyright © 2013 by the authors. All rights reserved.

A Language for Building Verified Software Components

Gregory Kulczycki[1], Murali Sitaraman[2], Joan Krone[3], Joseph E. Hollingsworth[4], William F. Ogden[5], Bruce W. Weide[5] , Paolo Bucci[5], Charles T. Cook[2], Svetlana Drachova[2], Blair Durkee[2], Heather Harton[6], Wayne Heym[5], Dustin Hoffman[5], Hampton Smith[2], Yu-Shan Sun[2], Aditi Tagore[5], Nighat Yasmin[7], and Diego Zaccai[5]

[1] Battelle Memorial Institute, Arlington, VA, USA
kulczyckig@battelle.org
[2] School of Computing, Clemson University, Clemson, SC 29634, USA
{murali, ctcook, sdracho, bdurkee, hamptos, yushans}@clemson.edu
[3] Mathematics and Computer Science, Denison University, Granville, OH 43023, USA
krone@denison.edu
[4] Computer Science, Indiana University Southeast, New Albany, IN 47150, USA
jholly@ius.edu
[5] Computer Science and Engineering, Ohio State University, Columbus, OH 43210, USA
{ogden, weide, bucci, heym, hoffmand, tagore, zaccai}@cse.ohio-state.edu
[6] Integrated Support Systems, Seneca, SC 29672, USA
hkeown@g.clemson.edu
[7] Computer Science, University of Mississippi, Oxford, MS 38677, USA
yasmin@clemson.edu

Abstract. Safe and secure reuse demands construction and use of verified reusable software components. Such verified components need much more than typical code for components in popular languages, such as C++ or Java. The components need to have formal specifications of behavior against which their implementations are verified. To be trusted, such verification must go beyond extensive testing and arguments of correctness, and must come with mechanized proofs. This paper discusses a realization of key elements of the conceptual idea of formal reasoning about software component behavior outlined in [1] (2000 ICSR proceedings). In the process, it summarizes central features of a language for building verified components must possess and a system that implements such a language. It explains that the language must include specifications as an integral constituent and must have clean semantics, which preclude unexpected side-effects and aliasing. The language must include mechanisms for writing reusable components that are amenable to verification, and consequently must also include an open-ended mechanism for adding arbitrarily sophisticated mathematical theories in order to specify large software components concisely. Because current programming languages lack these essential characteristics, the goal of verified components will remain unrealized unless the focus shifts to the design and development of a suitable language within which full verification is possible.

Keywords: assertive language, clean semantics, components, reuse, specification, and verification

1 Introduction

In order to achieve maturity as a field and to build safe and secure high assurance systems, software engineering must move from its current “cut-and-try” approach to a rigorous mathematically based system for engineering software. This engineering requires a language carefully designed to facilitate construction of verifiable and reusable software components and a verifying compiler—a compiler that generates code and also checks that code is correct. This is unarguably a grand challenge for the computing community [2].

This paper makes the following central contribution. It motivates and delineates the essential features of a language or framework for building verified components. It gives an example to illustrate how various features of the language affect component design, specification, implementation, and verification. Understanding the nature of a language for verified component development will help not just language designers, but also verified component developers in existing languages, informing them of potential pitfalls when one or more features of their language is in conflict with the goal of verification.

The rest of the paper is organized into the following sections. Section 2 outlines the essential features of a language for building verified components. Sections 3 to 6 of the paper use the examples from [1] to explain general language design principles of clean semantics, specification language, reusable mathematical theory development, and reusable components, respectively. To make these ideas concrete, the sections use the RESOLVE language and its supporting system [3] designed to support construction of verified software components. Space limitations preclude us from presenting more examples, so we refer an interested reader to various other sources [4, 5, 6]. The components and screenshots used in this paper can be accessed and attempted through the component finder (depicted in Figure 1) of the RESOLVE Web IDE available at www.cs.clemson.edu/group/resolve. Section 7 discusses the most related work and section 8 contains our conclusions.

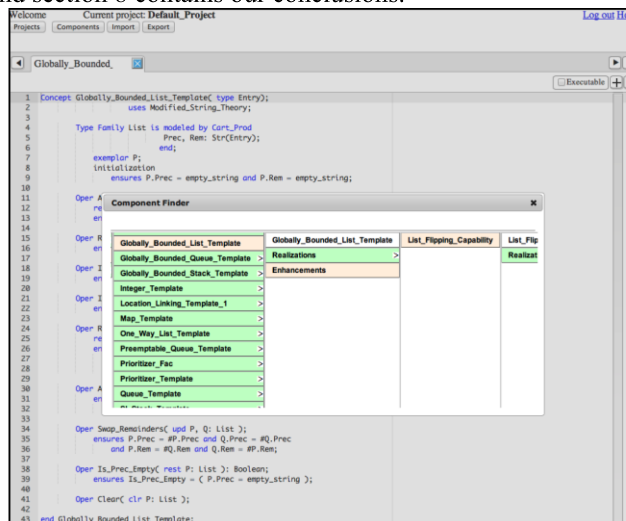

Fig. 1 – Component selection through the RESOLVE Web IDE

2 Essential Features of a Language for Verified Components

The essential features of a system for building verified software components must clearly include a language in which sophisticated, clean software can be written, and a specification system in which concise, precise intentions for the behavior of software components can be expressed. To insure the soundness of the verifying compiler, the specification mechanism and the programming mechanism must be fully coordinated in every detail, and about the only way to guarantee this is to integrate them into a single assertive programming language. The correctness objectives for the verification system and for the compiler can then be unified via a shared semantics for the language, and the all-too-common problem of incorrect behavior by seemingly verified software can be avoided. The need for a programming language within which specifications are an integral feature is the first of numerous indications that current programming languages are not adequate for meeting the grand challenge.

Another common problem occurs when verification is attempted in programming languages that lack clean semantics [7]. By clean semantics, we mean that all operations constructible in the language can only affect the objects to which they appear to have access. Typical cleanliness flaws arise from object aliasing and passing repeated arguments to an operation. The problem is that seemingly "verified" constituents may not behave correctly when employed in a larger system.

For a programming language to have clean semantics, the built-in data structures must be clean, and the composition mechanisms must preserve cleanliness. Constructing such a clean language by constraining an existing language to a sublanguage will essentially guarantee the creation of an unsatisfactory product, because the resulting language will be impractically weak. So a new language is required.

A major source of problems confronting verification is scale. Viewed from the specificational perspective, one such problem is that descriptions of the intended effects of programs would seemingly have to grow roughly in proportion to the size of the code. Given the limitation of human cognitive capacity, this is a very serious concern.

The solution to the coding side of this scaling problem is to provide modularization mechanisms that support a divide and conquer approach via componentization of software, with programming taking place using progressively more powerful components. An analogous approach is required on the specificational side, with descriptions of more powerful components being formulated in terms of more sophisticated theories. The net effect is that the collection of mathematical theories used in specifying software must remain open ended in order to support the growth of software driven by the rapidly increasing power of hardware. So an immediate corollary is that machinery for developing mathematical theories must be a third constituent of the language for specifying software that we want to program.

Even with well-conceived program verification machinery, the cost of evolving poorly structured software into correct software is bound to remain prohibitively high. One of the primary strategies used by more mature engineering disciplines for achieving sound products at reasonable cost is to rely upon a comparatively small collection of highly reusable components, and this must surely be an approach that is strongly supported by a language for software verification.

There are several key features that the machinery provided for generating reusable components must have. Certainly it must exhibit the clean semantics mentioned above, since modular understanding is always essential to reuse. Second, it must provide the potential for a high degree of genericity, since keeping catalogues of reusable components to an intellectually manageable size is important. Third, it must provide an interfacing mechanism for presenting the object types and operations together with their abstract specifications, since information hiding is critical to keeping the specifications of higher-level code as simple as possible. Fourth, it must support the development of alternative implementations of an abstract interface, since different implementations of the same functionality are necessary to meet different performance goals, and if a component interface does achieve the desired degree of reusability, then it must be possible in a large system to deploy it in numerous places where varying performance requirements hold.

Reusable components are the setting in which the specificational simplification derived from changing to more sophisticated mathematical theories frequently occurs. So part of the machinery in a component implementation must provide the specification of a correspondence relation that properly matches the behavior of the entities at the implementation level with functionality prescribed for the more abstract entities presented by the component's external interface. Performance specification and verification capability is also essential for a satisfactory software verification system, and accordingly, component implementation machinery must provide for translating this information up to the external interface.

3 Clean Semantics

Correct reasoning about software, both formal and informal, is critically dependent on "separation of concerns." If a piece of code appears to be working on only a small portion of the overall state space, then any efficient verification system must be safe in restricting its attention exclusively to the code's effect on that subspace. Languages that restrict the effects of each programming construct to just the objects that are syntactically targeted by the construct are said to have clean semantics [3], so a language with clean semantics is an important requirement if verification is to succeed.

The biggest impediment to this separation of concerns and clean semantics in a language is unconstrained aliasing. Aliasing forces programmers (and verifying compilers) to constantly account for the added levels of indirection present in order to reason soundly about their code. For example, in the array assignment A[i] := 47, the array A remains unchanged except for its i-th element. As no other variables are involved in the assignment, we would like to know that no other part of the program state has changed. If the language has clean semantics, we are assured of this. However, if the language permits a different variable B to be aliased to A, we must understand that B's value has also changed. For a verification system to remain sound in an environment with unconstrained aliasing, it must include an implicit, omnipresent, and sophisticated heap variable in its model of the program state, causing the reasoning system to be perilously—and unnecessarily—complex.

To clarify the idea of clean semantics further, consider the example in [1], where the verification of a piece of code to reverse a linked list, such as the one reproduced in Figure 2 (from [1]) is discussed. The client or user code for list reversal can be layered using primary operations from a List component, such as insert and remove. Verification of these primary operations within a List component implementation indeed requires non-trivial reasoning about references and aliasing, which can be shown formally, but not yet mechanically [8, 9, 10]. On the other hand, in a clean language, reasoning about most routine user code such as list reversal would be straightforward. Specifically, a language suitable for verification must be designed to encapsulate pointers within components that are carefully engineered to present clean, efficient interfaces to programmers. Such interfaces are the topic of the next section.

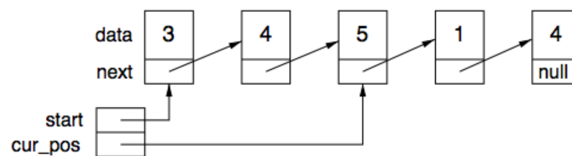

Fig. 2 – A typical singly linked list representation

As reference copying is the main cause of unconstrained aliasing, it follows that to support clean semantics without sacrificing efficiency, the language must also support object movement mechanisms that avoid reference copying, which can be accomplished via swapping or transfer [11, 12]. It must also provide a way to pass parameters to procedures such that repeated arguments do not introduce aliasing [7].

4 Language Support for Specifications

If systems that do not share a common design are combined to write and verify software, the slightest of inconsistencies in their semantics could easily vitiate apparent correctness results. Consequently, we need one language that treats the development of software systems as an integrated whole. In particular, software's specifications should be viewed as an essential part of the software, and not as an add-on sideshow that might or might not describe the actual code.

A clean specification of List abstraction (devoid of complications due to pointers) is given in [1] and an updated, verification-friendly version of that specification, shown in the background in Figure 1 can be found at the RESOLVE Web IDE. In the specification, a list is conceptualized as an ordered pair: a mathematical string of entries that precede the insertion point (denoted by *Prec*) and a string of entries that remain past the insertion point (denoted by *Rem*). The abstraction for the linked list in Figure 1 is just the ordered pair (<3,4,5>, <1,4>). In general, all types need syntactic slots where software engineers can express mathematical conceptualizations for the type's abstraction and all operations need syntactic slots to specify the pre and post conditions (or requires and ensures clauses) that describe the situations in which an operation can be invoked and the outcomes that it is designed to produce.

In Figure 3, a formal specification of a list reversal operation, named Flip_Rem is shown as a screenshot from the IDE; this operation is an enhancement (or extension) to the list component. In this specification, Reverse is a mathematical function that reverses a string; its formal definition is given in the next section. It is specified to take a list, such as (<>, <1,2, 3,4>) and produce (<4,3, 2,1>,<>).

Projects | Components | Import | Export

Globally_Bounded_ | List_Flipping_Capa | Recursive_List_Flip

Save | Rename | Delete

```
1  Enhancement List_Flipping_Capability for Globally_Bounded_List_Template;
2      Operation Flip_Rem (updates P: List);
3          requires P.Prec = empty_string;
4          ensures P.Prec = Reverse(#P.Rem) and P.Rem = empty_string;
5  end List_Flipping_Capability;
```

Fig. 3 – Specification of a List operation to reverse a list

The meaning of correctness of an implementation of Flip_Rem depends on its specification and the specifications of operations it uses; i.e., the same code may be correct or wrong, underscoring that specifications should be treated as an integral part. Stated more formally, pre and post conditions can produce effects involving two special semantic states: a vacuously correct state, VC, and a manifestly wrong state, MW. If, for example, some code attempts to invoke an operation in a state that does not meet the operation's precondition, then the resulting state is MW. Similarly if the code for an operation does not meet its post condition, then the outcome is also MW. The VC state is introduced when the code for an operation is started in a state that does not meet its pre condition. A program is semantically correct only if under no circumstances can it produce the MW state. In such an integrated approach, the potential for a verification system to be unsound is vastly reduced.

A compiler is only going to be capable of verifying the correctness of assertive code if that code includes sufficient hints in the form of justificational specifications, which are provided by the software engineer, to make intermediate deductions "obvious." Besides pre and post conditions for operations, the language must support invariants for its looping constructs, invariants and abstraction relations for its component modularizing mechanism, etc. To account for tricky code, constructs for adding specificational claims as supplemental hints will be necessary. The semantics of such specificational constructs can also be described easily using the two semantic states described above. Other such constructs will be required to account for the ordinal valued progress metrics that are required to ensure the termination of loops and recursive procedures.

While we do not discuss performance-verified components in this paper (because we do not yet have an automation of those ideas), no verification system would be considered adequate if it did not account for performance. So there must be slots for integer valued expressions used to specify the memory displacements required by objects and transitorily by operations. The slots that account for the durations of operations, loops, etc., will hold real-number-valued expressions, so it is clear that the verification system must deal with a variety of number systems before it even begins to work with anything introduced to specify particular components.

5 Reusable Mathematical Theory Constituent

In the thirty or so years since the De Millo, Lipton, and Perlis critique of the prospects for program verification [13], our understanding of automated proof systems has made modest progress, our computers' speed has made dazzling progress, and the social "proof" process for verifying the correctness of general mathematics has changed hardly at all. Consequently, results found in the mathematical literature still somewhat resemble software in that the parts that have been heavily used are pretty much correct, while both the rarely used and newly developed parts are more suspect.

There is no particular reason to believe that the most appropriate theories for specifying the full compass of software applications will inevitably lie within the well-worked parts of mathematics, so the reliability of the general software verification process becomes quite suspect unless it rests on a firmer foundation than citations into the mathematical literature. In short, the mathematics used in software specification and verification must be industrial strength rather than craftsman formulated. A system for developing, checking, and cataloguing mathematical theories then becomes an essential component of a software verification system. For example, while a theory of mathematical strings could be codified in a specification language and employed for effective verification [4], in general, the language should make it possible to define and use new theories without modifying and recompiling the verifier.

Brief reflection reveals that the notations and results developed in mathematical theories are not independent of program specifications or of the verification process. In fact, it is essential that the specifications and the mathematical theories share a common semantics, and that makes the mathematical theories and the syntactic constructs used in their formulation a part of the language used for programming and specification.

Several ideas from reusable component engineering are also appropriate for structuring mathematical theories. The first is separation of concerns. A client using a theory to formulate specifications only needs a summary or précis of the definitions and results (theorems) for that theory, but not anything about proofs for the results, so the précis should be in a separate syntactic unit from the proofs. This is analogous to separating interfaces and implementations in the programming constituent of the language in that a client of an interface in both cases only needs to know that the supporting unit exists but can safely ignore the messy details that are left to another specialist.

A second such idea is reuse itself. Well-considered and well-developed mathematical theories are appropriate for a variety of specifications, and the cost of their formulation and proof can be amortized over all these uses.

Shown in Figure 4 below is a portion of a précis (interface) for a reusable String Theory. The inductive definition of Reverse in the précis is preceded by formal definitions of the symbols used in Reverse (these are not shown), including Str, empty_string, ext (extension of a string with an entry), o (concatenation), |•| (string length) and <•> (string containing one element).

The form and degree of development of libraries of theories is a primary determiner of whether the fully automatic verification of particular software can succeed. Results provided by the theories are to be proved off-line by mathematically

proficient specialists, so they can be arbitrarily difficult. It is highly unlikely that a verifying compiler is ever going to be able to prove anything but easy stuff. So it is up to the theory developers and the software specifiers to leave only a narrow gap between the strong results in the theory development and the specifications and specificational hints in the software. This narrow gap is what gives hope for automated software verification. Given this objective, these theory developments may contain some results that are a bit too obvious for a traditional mathematical development, but are an essential aid for a verifying compiler.

```
Theory String_Theory;
    uses Number_Theory, …;
    …
    Inductive Definition Reverse(s: Str(Gamma)): Str(Gamma) is
        (i)  Reverse(empty_string) = empty_string;
        (ii) For all x: Gamma, Reverse(ext(s, x)) = <x> o Reverse(s);
    …
    Theorem S7: For all u, v:Str(Gamma), Reverse(u o v) =
        Reverse(v) o Reverse(u);
end String_Theory;
```

Fig. 4 – String_Theory Précis

Making the verification of production software routine depends on a taxonomic thesis about how software engineers create software that they "know" is correct. The thesis is that most of such code is straightforward and it is plain to see that it is correct [14]. The remaining not-so-obvious parts are separable from the rest, and certainty of the correctness of each such part is developed through a serious individual process of abstract reasoning. If this thesis is correct, then a software verification system can achieve its objectives using two qualitatively different subsystems. The first addresses the not-so-obvious and is the general mathematics subsystem that handles theory and proof modules. The second is a code justification checker that examines the specifications embedded in code to determine whether they are "obviously" correct, given the specifications and annotations in the code and the definitions and theorems developed in the supporting theories.

A language for developing verifiable software must be simultaneously cognizant of the limits of mechanization and of the needs of language users. For example, both because mathematicians generally will be needed to develop the non-trivial proofs upon which some software relies and because traditionally educated software engineers will be required to read and write specifications, it is essential that the software verification system present theories and specifications in standard mathematical notation.

6 Verified Reusable Components

Reusable components would seem to be inextricably connected to verification. If components are to be deployed across a large base of installed software, then it is essential that they be correct, since the cost assessment associated with any residual

errors contains a huge multiplier. By the same token, when highly reusable components are fully specified and verified, the cost can be amortized over a large base of usage.

Yet nurturing this symbiotic relationship does not appear to have been a serious concern. Notions such as reuse via inheritance are widespread, when inheritance clearly exhibits terrible coupling properties and thoroughly undermines modular verification.

Verification supportive component interfacing machinery must cleanly decouple the implementations of components from their deployments. This involves interfacing mechanisms for clearly identifying all the fixed and parametric entities that are employed by a component, the properties the parameters must possess, and the properties of the objects and operations provided by the component that can be relied upon in all deployments.

Because maximal generality is essential, the component parameterization mechanism must be designed to support full situational flexibility. For example, a parameterization mechanism that allows the passing of object types and their operations only as whole components forces stamp coupling in situations when only a few of the operations are needed, and this clearly restricts the number of situations in which such components could be employed. A more sophisticated reuse supportive mechanism would allow fine-grained parameter passing for components. Such an example is discussed in [6].

The principal goal of the decoupling provided by component interfacing machinery is information hiding, and the abstract specifications provided for exported objects and operations provide the cover stories behind which a myriad of sordid implementation details can be hidden [15]. These "abstract" specifications become the concrete specifications when a component is used to construct higher level software, so keeping these specifications as simple as possible (i.e., keeping the gap narrow) is essential if the verification of such higher level software is to remain tractable. This decoupling is the motivation for the abstract specification of List component. Using that specification, it becomes possible to cleanly verify the realization (code) in Figure 5 is correct with respect to the specification of the enhancement operation Flip_Rem, presented earlier in Figure 3.

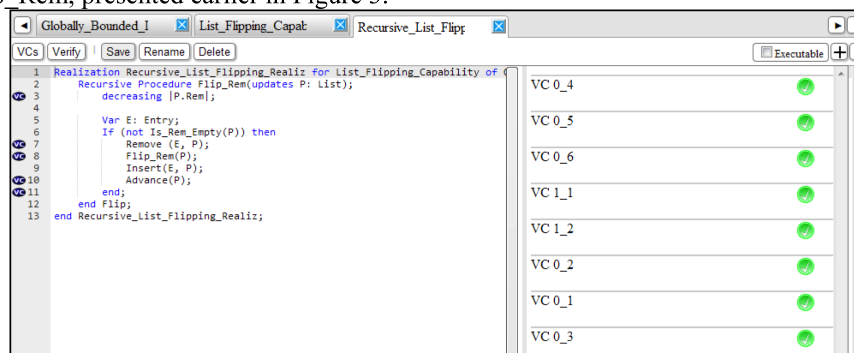

Fig. 5 – Verified client implementation of Flip_Rem

The code in Figure 5 is the same as the one presented in [1], except that this one is mechanically verified. The syntactic slot for the decreasing clause in this recursive

procedure makes it possible for a software engineer to annotate the code and enable automatic proof of termination. Notice that in Figure 5 the five blue ovals down the left hand side containing the letters “VC”. VC stands for verification condition, and for the operation “Flip_Rem” the RESOLVE verifier has analyzed the code along with specifications and automatically generated all the verification conditions (detailed in [1]) required for proving the correctness of Flip_Rem code. Clicking the VCs button seen at the top of the screen shot generates the VCs. Figure 6 lists the VCs associated with the last blue oval. These four VCs (a product of the two conjuncts in the ensures clause of Flip_Rem and the two paths through the code) have to do with proving the ensures clause for Flip_Rem. One example VC, VC 0_5 is shown in Figure 7. It stems from the specifications of operations involved and includes string notations |•|, <•>, DeString, where DeString(<x>) is x, and Prt_Btwn (m, n, s), that is a substring of s between m and n. To discharge this VC, the verifier employs a variety of theorems, including the Reversal theorem “S7” from Figure 4.

```
13        end;
VC: 0_5: Ensures Clause of Flip_Rem , If "if" condition at
Recursive_List_Flipping_Realiz.rb(8) is true:
Recursive_List_Flipping_Realiz.rb(13)

VC: 0_6: Ensures Clause of Flip_Rem , If "if" condition at
Recursive_List_Flipping_Realiz.rb(8) is true:
Recursive_List_Flipping_Realiz.rb(13)

VC: 1_1: Ensures Clause of Flip_Rem , If "if" condition at
Recursive_List_Flipping_Realiz.rb(8) is false:
Recursive_List_Flipping_Realiz.rb(13)

VC: 1_2: Ensures Clause of Flip_Rem , If "if" condition at
Recursive_List_Flipping_Realiz.rb(8) is false:
Recursive_List_Flipping_Realiz.rb(13)
```

Fig. 6 – VCs associated with establishing the ensures clause of Flip_Rem

Support for a verifier is integrated with the RESOLVE Web IDE, and is accessed by clicking the Verify button. Many VCs for our components can be automatically proved, but not all since theorem proving is still an open research topic. Details of the Web IDE can be found in [6] and of the verifier in [5].

```
Globally_Bounded_   List_Flipping_Capa   Recursive_List_Flip
VCs  Verify  Save  Rename  Delete                    Executable

1  Realization Recursive_List_Flipping_Realiz for
2      List_Flipping_Capability of Globally_Bounded_List_Template;
3
4      Recursive Procedure Flip_Rem(updates P: List);
5          decreasing |P.Rem|;
6
7          Var E: Entry;
8          If (not Is_Rem_Empty(P)) then
9              Remove (E, P);
10             Flip_Rem(P);
11             Insert(E, P);
12             Advance(P);
13         end;
14     end Flip;
15 end Recursive_List_Flipping_Realiz;
16
17
```

```
VC 0_5

Ensures Clause of Flip_Rem , If "if" condition at
Recursive_List_Flipping_Realiz.rb(8) is true:
Recursive_List_Flipping_Realiz.rb(13)

Goal:

    (Reverse(Prt_Btwn(1, |P.Rem|, P.Rem)) o Prt_Btwn(0,
    1, (<DeString(Prt_Btwn(0, 1, P.Rem))> o
    empty_string))) = Reverse(P.Rem)

Given:

    1. (min_int <= 0)
    2. (0 < max_int)
    3. (Last_Char_Num > 0)
    4. P.Prec = empty_string
    5. P_val = |P.Rem|
    6. not(P.Rem = empty_string)
```

Fig. 7 – Automatically generated verification condition VC 0_5 for Flip_Rem

Two properties of a verification system are the keys to allowing software engineers to achieve such simplicity in their component specifications. The first is that the system has clean semantics, for the reasons previously discussed. The second is for the system to provide semantics that are *rich*. By this we mean that if a software engineer can come up with any mathematical system in terms of which he can provide a complete, accurate explanation of the behavior of a component, then the verification system should fully support his use of that mathematical system in his specification of the component. The idea is that a verification system that has rich semantics cannot preclude the development of component specifications that are the simplest possible. For example, forcing a programmer/specifier to express every component description in terms of a single theory, such as set theory, would lead to possibly accurate, but unreadable specifications.

We conjecture that there is no practical way to distinguish the mathematical theories that might ever be useful across the ever expanding sphere of software applications. This is why we expect that the compass of the mathematical theory constituent of a software verification system would be coextensive with the domain of general mathematical theories. An interesting corollary is that, semantically, a verification system would have to support programming in arbitrary mathematical domains because, when any component is deployed, the code that uses it will be verified against a semantics based upon that component's mathematical model.

The use of different mathematical theories to provide nicer cover stories for implementations using specifications based on more complex theories means that, when the system is verifying implementations, it is going to need programmer supplied specifications for correspondence relations between the two theories so that pre and post conditions written in the cover story theories make sense in terms of the theories employed at the implementation code level. If the mathematics in the cover story really is simpler, then the extra verificational complexity introduced here will be recompensed many times over – especially if the component is highly reusable.

It is important to note that the implementation in Figure 5 and its verification are independent of how exactly lists are implemented, i.e., they are both reusable. This is another major benefit of information hiding in that it allows software designers to develop alternative implementations that all match the abstract cover story presented by a single component interface (List component interface, in this case) but that differ in structural details and performance characteristics. If a verification system supports plug-compatible interchange in such situations, it can achieve significant savings because the functional reasoning about any deployment of the component depended entirely on the specifications in the component's interface and not at all on details in an implementation. Of course the details of an implementation do dictate what correspondence relation is needed to make them match with the cover story, so each implementation module is going to need its own correspondence specification.

Because many of today's component libraries are cluttered with similar but not quite compatible entries, there is also an advantage for a language to offer strong support for developing broadly unifying component interfaces that bring together a multitude of alternative implementations into a relatively few standard and highly reusable components. These unifying component interfaces would be much easier for programmers to understand and employ. The payoff for verification from the increased use of these components is obvious.

Efficiency is also a critical consideration for verifiable software in general and for reusable components in particular. As noted, the programming constituent of a verification system should not be so bereft of built-in capabilities that it prevents software engineers from developing software that is essentially as efficient as that developed in current programming languages.

When it comes to components, there is a concern that specificational deficiencies will prevent software engineers charged with component implementation from achieving full efficiency. For example, with many sorts of components there is an operation to return from a container object an item meeting certain criteria. From a client perspective, the container may well hold several items all meeting his criteria and it is a matter of complete indifference which is returned, so the natural input/output specification is relational. If however the verification system only handles functional specifications, then the operation's post condition will have to be made unnecessarily exacting (which may even involve complicating the model of the container). The result would be over specification and the exclusion of potential implementations having better performance in either time or space. Clearly, a verification system needs to support relational specifications.

In component based software, as we have noted, "abstract" specifications for a component become concrete specifications where that component is deployed, and when component specifications are relational, this means that the semantics of the programming component of a verification system must become relational. Because traditional denotational semantics assume that the semantics will be functional (i.e., for an input state, produce a single output state), the semantics of verification must be established on a new basis. This is quite doable, but it does involve moving the fixed point result from the classical continuous function based argument to one involving taking limits over ordinals.

7 Safety, Security, and Related Work

The achievement of verifiable reusable software components is an important one in its own right as a revolutionary advancement in software engineering. However, the goal takes on an even greater significance when viewed from the perspective of high-assurance systems, where safety and security are critical properties. Formal methods have always been in demand in safety-critical systems [16]. And formal techniques are becoming increasingly relevant in cyber-security. In a 2010 US-DoD-sponsored JASON report on the science of cyber-security, formal methods was highlighted as a key research area that could provide scientific principles for the domain of cyber-security [17]. Security and safety are closely related, particularly when it comes to cyber-physical systems such as vehicles. The increasingly automated control systems in automobiles coupled with demonstrations of how easy it is to hack a typical automobile are a cause for concern [18].

The Common Criteria—a widely recognized international standard for assurance evaluation—includes certification levels ranging from EAL1 (functionally tested) up to EAL7 (formally verified) [19]. At the time of this writing, only a handful of systems have achieved EAL7, and those that have do not include extensive software

portions. Tools used to verify such systems include model checkers, which do not scale well for software using complex data types, and interactive theorem provers, which require significant expertise and human interaction to use.

Though current industrial-strength tools for verification all have limitations, impressive results have been achieved. For example, the ACL2 interactive theorem prover has been used to certify a microprocessor with 200 instructions at EAL7 and an real-time operating system separation kernel (hypervisor) at EAL6+ [20]. The SPIN model checker has been used to verify selected algorithms for various NASA spacecraft, including the Mars Rovers. It was also employed in the investigation of the control software for Toyota Camry breaking systems [21]. The Static Driver Verifier is a tool in the Windows Driver Development Kit that uses the SLAM verifier (a sophisticated model checker) to statically verify whether a kernel-mode driver implementation interacts properly with the Windows operating system [22]. SeL4 is a general-purpose operating system kernel that was verified using the Isabelle/HOL theorem-proving environment [23]. The verification effort took nearly eleven person-years to complete, though the kernel itself is comprised of only about 10,000 lines of code.

Despite these efforts, scalable verification will only be achieved with tools and techniques that are fully integrated into the software engineering process—such as the verifying compiler for the language we describe. Several efforts have been made at designing a verification system for new or existing languages. Functional examples of software verification systems include Isabelle/HOL derivatives [24], PVS [25], and ACL2 [26]. Functional languages do provide clean semantics, but those described here do not provide a high degree of modularity or abstraction, as most object-based languages do. Also, these languages are dependent on interactive rather than automated theorem proving. Some verification systems begin with the specification and arrive at the code through a process of refinement. Kestrel Specware is a good example of a system that uses this process [35]. Automatic code generation may be possible using this technique when restricted to a specific domain, but in general the refinement process is also interactive.

Imperative examples of verification systems include VCC for C [27]; JML (Java Modeling Language) for Java [28]; Spec#, a superset of C# [29]; Dafny, an object-oriented language designed for verification [30], SPARK, an Ada-based language [31], and RESOLVE, the language designed by our research group according to the principles laid out in this paper [3]. Of these imperative languages, only Dafny and RESOLVE are languages that have been built from the ground up to support verification. Dafny is an interesting case in that even though it does not have clean semantics, considerable effort has been made to ensure that reasoning about aliasing remains sound [30]. We know that this decision complicates the verification conditions generated by the system. Dafny also couples implementations with specifications, posing potential problems for modularity [33]. Whether this coupling is due wholly or in part to a lack of clean semantics is still an open question. Jahob has similar characteristics to Dafny but works on a subset of Java [32]. A helpful comparison of many of the verification systems mentioned here can be found in [34].

8 Conclusions

To meet the challenge of building verified software components, a verification-driven language design is necessary. Characteristics of such a language do not match those of currently popular languages. Recognizing this means that the fettered, de facto language redesign activity that is language subsetting is not a viable strategy.

In effect, language development constitutes the bulk of the verified software challenge because, as we have noted, the overarching soundness requirement means that the language must include mechanisms for both code specification and mathematical development of the theories used in these specifications.

A verifying compiler then must process all these language constituents, so that it effectively includes components that perform the traditional syntax checking and code generation augmented by those that check the results in the mathematical theories and that check the correctness of claims made in the program specifications. As mentioned, the language must be cleanly modular so that separate compilation of supporting theory units, software components, etc. can be standard operating procedure. A more subtle point is that the verification system must maintain complete control of previously compiled modules if the integrity of verifications is to be maintained.

Succeeding in the goal of building verified software components still will not mean that all the software the compiler processes is absolutely correct because that software may not have been properly specified to meet the objectives of the real world system in which it is to be embedded. The analysis of requirements and the specification of real world systems with embedded software will almost certainly require even more complex techniques than those for specifying and verifying correctness within the relatively pristine world of imperative programming. However, complete specification and verification of software components would imply a clear separation of concerns. Questions about the correctness of systems in which software is embedded could be fully addressed by looking only at the specifications for that software, with absolutely no consideration of coding details, and questions about the correctness of software could be fully addressed with absolutely no consideration of the systems into which it is to be embedded.

Developing a verifying compiler and building verified components would certainly represent a major advance for our field, but the challenge will only be met when the realities of what is involved are squarely faced. Among these realities is the need to educate the next generation of software engineering workforce on the principles of verified software component construction. We have made significant progress in this direction [36]. Students at about a dozen US institutions have been introduced to building reusable components according to specifications and establishing component correctness using the web IDE.

Acknowledgments. We wish to thank our research groups for their contributions over the decades to the ideas discussed here. Various ideas presented here were developed under United States National Science Foundation grants CCF-0811748, CCF-1161916, DUE-1022191, and DUE-1022941.

References

1. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S., and Hollingsworth, J.E., Reasoning About Software-Component Behavior. In Frakes, W.B., ed., Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse), Springer-Verlag LNCS 1844, 266-283 (2000)
2. Hoare, T., The Verifying Compiler, a Grand Challenge for Computing Research. JACM 50, 1 63-69 (2003)
3. Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, B., Frazier, D., Friedman, H.M., Harton, H.K., Heym, W., Kirschenbaum, J., Krone, J., Smith, H., and Weide, B.W., Building a Push-Button RESOLVE Verifier: Progress and Challenges. Formal Aspects of Computing 23, 5, 607-626 (2011)
4. Adcock, B., Working Towards the Verified Software Process. Ph. D. thesis, Computer Science and Engineering, The Ohio State University (2010)
5. Harton, H.K., Mechanical and Modular Verification Condition Generation for Object-Based Software. Ph. D. Dissertation, Clemson University, 305 pages (2011)
6. Cook, C.T., Harton, H.K., Smith, H., Sitaraman M., Specification engineering and modular verification using a web-integrated verifying compiler. In Proceedings of the International Conference on Software Engineering (ICSE 2012), 1379-1382 (2012)
7. Kulczycki, G., Direct Reasoning. Ph. D. Dissertation, Clemson University, 183 pages (2004)
8. Jensen, J.B., Birkedal, L., Sestoft, P., Modular verification of linked lists with views via separation logic. In Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs (FTFJP '10), 7 pages, (2010)
9. Zee, K., Kuncak V., Rinard M., Full functional verification of linked data structures. In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM Press, 349-361 (2008)
10. Kulczycki, G., Smith, H., Harton, H., Sitaraman, M., Ogden, W.F., Hollingsworth, J.E., The Location Linking Concept: A Basis for Verification of Code Using Pointers. VSTTE'12 Proceedings of the 4th international conference on Verified Software: theories, tools, experiments, 34-49 (2012)
11. Harms, D.E., and Weide, B.W., Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Transactions on Software Engineering 17, 5 (May 1991), 424-435 (1991)
12. Weide, B.W., Heym, W.D., Specification and Verification with References. Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems, (2001)
13. DeMillo, Richard A., Lipton, Richard J., Perlis, Alan J.: Social Processes and Proofs of Theorems and Programs. Comm. ACM 22, 5, 271-280 (1979)
14. Kirschenbaum, J., Adcock B., Bronish, D., Smith, H., Harton, H., Sitaraman M., Weide, B.W., Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping? In Proceedings 11th International Conference on Software Reuse, Springer LNCS 5791, 31-40 (2009)
15. Parnas, D.L.: A Technique for Software Module Specification with Examples. Comm. ACM 15, 5, 330–336 (1972)
16. NASA Langley Formal Methods Program. Why is formal methods necessary. http://shemesh.larc.nasa.gov/fm/fm-why-new.html
17. McMorrow, D., Science of Cyber-Security. In Cyberwar Resources Guide, Item #98, http://www.projectcyw-d.org/resources/items/show/98 (2010)

18. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., Comprehensive Experimental Analyses of Automotive Attack Surfaces. USENIX Security (2011)
19. The Common Criteria Portal. http://www.commoncriteriaportal.org/
20. Hardin, D.S. ed., Design and Verification of Microprocessor Systems for High-Assurance Applications. Springer (2010)
21. Holzmann, G.J., The Spin Model Checker. Addison Wesley (2005) http://spinroot.com/spin/whatispin.html
22. Ball, T., Levin, V., and Rajamani, S.K., A Decade of Software Model Checking with SLAM. Communications of the ACM. (July 2011)
23. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S., seL4: Formal Verification of an OS Kernel. In Proceedings ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). ACM, 207-220. (2009)
24. Nipkow, T., Wenzel, M., and Paulson, L.C., Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer-Verlag, Berlin, Heidelberg. (2002)
25. Crow, J., Owre, S., Rushby, J., Shankar, N., and Srivas, M., A Tutorial Introduction to PVS. In Proceedings: Workshop on Industrial-Strength Formal Specification Techniques. (1995)
26. Kaufmann, M., Manolios, P., and Moore, J.S., ACL2: Computer-Aided Reasoning: An Approach. (2011)
27. VCC: A Verifier for Concurrent C. http://research.microsoft.com/en-us/projects/vcc/
28. Chalin, P., Kiniry, J.R., Leavens, G.T., and Poll, E. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, pages 342-363. Volume 4111 of Lecture Notes in Computer Science, Springer Verlag, 2006.
29. Barnett, M., Leino, R.K.M., and Schulte, W. The Spec# programming system: An overview. In CASSIS 2004, LNCS vol. 3362, Springer, 2004.
30. Leino, K.R.M., Dafny: an automatic program verifier for functional correctness. In Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning (LPAR'10). Springer-Verlag, 348-370. (2010)
31. Jennings, T.J., SPARK: the libre language and toolset for high-assurance software engineering. In: SIGAda '09 Proceedings of the ACM SIGAda annual international conference on Ada and related technologies. (2009)
32. Zee, K., Kuncak, V., and Rinard, M. An integrated proof language for imperative programs. In ACM Conf. Programming Language Design and Implementation (PLDI), 2009.
33. Bronish, D. and Weide, B.W., A Review of Verification Benchmark Solutions Using Dafny. In Proceedings of VSTTE 2010 (Verified Software: Theories, Tools, and Experiments) Workshops (2010)
34. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B. Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., and Weiss, B. The 1st Verified Software Competition: Experience Report. In FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Springer-Verlag LNCS 6664, 154-168 (2011).
35. Smith, D.R., Generating Programs plus Proofs by Refinement. In Proceedings Verified Software: Theories, Tools, Experiments. Springer-Verlag LNCS 4171, 182-188 (2008).
36. Sitaraman, M., Hallstrom, J. O., White, J., Drachova, S., Harton, H. K., Leonard, D. P., Krone, J., Pak, R., Engaging students in specification and reasoning: "hands-on" experimentation and evaluation. In Proceedings ACM ITiCSE, 50-54 (2009).