Integrated Reasoning and Proof Choice Point Selection in the Jahob System (Mechanisms for Program Survival)

Martin Rinard

Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
32 Vassar Street, 32-G744
Cambridge, Massachusetts 02139

Abstract. In recent years researchers have developed a wide range of powerful automated reasoning systems. We have leveraged these systems to build Jahob, a program specification, analysis, and verification system. In contrast to many such systems, which use a monolithic reasoning approach, Jahob provides a general *integrated reasoning* framework, which enables multiple automated reasoning systems to work together to prove the desired program correctness properties.

We have used Jahob to prove the full functional correctness of a collection of linked data structure implementations. The automated reasoning systems are able to automatically perform the vast majority of the reasoning steps required for this verification. But there are some complex verification conditions that they fail to prove. We have therefore developed a proof language, integrated into the underlying imperative Java programming language, that developers can use to control key choice points in the proof search space. Once the developer has resolved these choice points, the automated reasoning systems are able to complete the verification. This approach appropriately leverages both the developer's insight into the high-level structure of the proof and the ability of the automated reasoning systems to perform the mechanical steps required to prove the verification conditions.

Building on Jahob's success with this challenging program verification problem, we contemplate the possibility of verifying the complete absence of fatal errors in large software systems. We envision combining simple techniques that analyze the vast majority of the program with heavyweight techniques that analyze those more sophisticated parts of the program that may require arbitrarily sophisticated reasoning. Modularity mechanisms such as abstract data types enable the sound division of the program for this purpose. The goal is not a completely correct program, but a program that can survive any remaining errors to continue to provide acceptable service.

1 Introduction

Data structure consistency is a critical program correctness property. Indeed, it is often directly related to the meaningful survival of the program. As long as a program preserves the integrity of its core data structures, it is usually able to execute through errors to continue to provide acceptable (although not necessarily perfect) service to its users [1–7].

We have developed a general program specification and verification system, Jahob, and used Jahob to verify, for the first time, the full functional correctness of a collection of linked data structure implementations [8–10]. This verification constitutes a key step towards the goal of statically ensuring data structure consistency — part of the verification process is ensuring that individual data structure implementations preserve invariants that capture key internal data structure consistency constraints.

1.1 Integrated Reasoning

To verify the full functional correctness of linked data structure implementations, Jahob must work with sophisticated properties such as transitive closure, lambda abstraction, quantified invariants, and numerical relationships involving the sizes of various data structure components. The diversity and complexity of the resulting verification conditions makes the use of a single monolithic reasoning system counterproductive. Instead, Jahob uses *integrated reasoning* — it implements a general framework that enables arbitrary reasoning systems to interoperate to prove the complex verification conditions that arise in this context [8, 10]. The success of integrated reasoning depends on two techniques:

- **Splitting:** Jahob splits verification conditions into equivalent conjunctions of subformulas and processes each subformula independently. Splitting enables Jahob to take a single formula that requires many kinds of reasoning, divide the formula up into subformulas (each of which requires only a single kind of reasoning), then apply multiple different reasoning systems as appropriate to solve the subformulas. The result is that Jahob is able to leverage the combined power of an arbitrary number of reasoning systems to solve complex formulas.
- **Formula Approximation:** In general, each reasoning system will have its own restrictions on the set of formulas that it will accept as input. Several formula approximation techniques make it possible to successfully deploy a diverse set of reasoning systems together within a single unified reasoning framework. These approximation techniques accept higher-order logic formulas and create equivalent or semantically stronger formulas accepted by specialized decision procedures, provers, and other automated reasoning systems.

  Our approximation techniques rewrite equalities over complex types such as functions, apply beta reduction, and express set operations using first-order quantification. They also soundly approximate constructs not directly

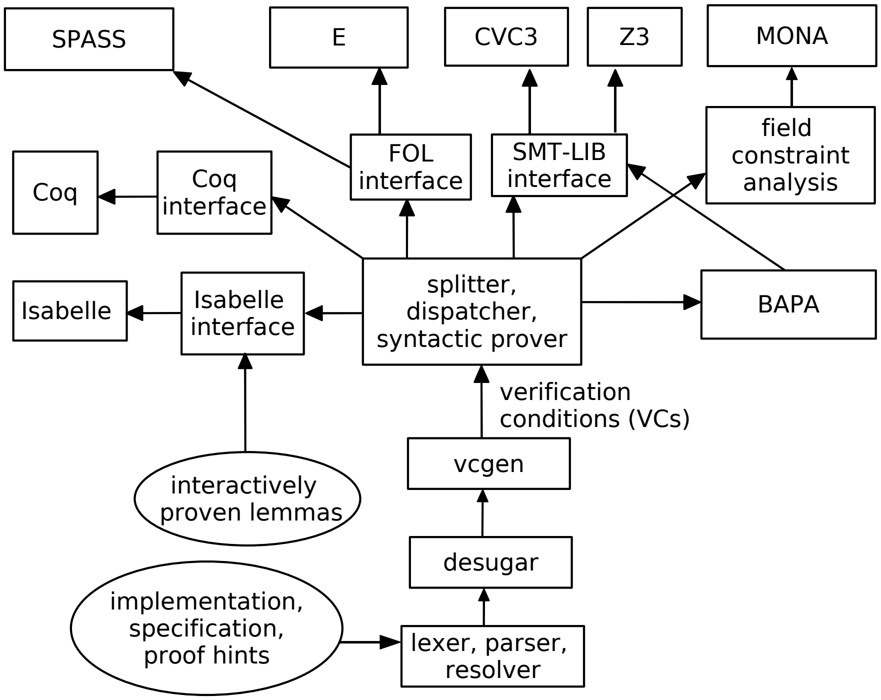
SPASS
E
CVC3
Z3
MONA
Coq
Coq interface
FOL interface
SMT-LIB interface
field constraint analysis
Isabelle
Isabelle interface
splitter, dispatcher, syntactic prover
BAPA
verification conditions (VCs)
vcgen
interactively proven lemmas
desugar
implementation, specification, proof hints
lexer, parser, resolver

Fig. 1. Integrated Reasoning in the Jahob System

supported by a given specialized reasoning system, typically by replacing problematic constructs with logically stronger and simpler approximations.

Together, these techniques make it possible to productively apply arbitrary collections of specialized reasoning systems to complex higher-order logic formulas. Jahob contains a simple syntactic prover, interfaces to first-order provers (SPASS [11] and E [12]), an interface to SMT provers (CVC3 [13, 14] and Z3 [15, 16]), an interface to MONA [17, 18], an interface to the BAPA decision procedure [19, 20], and interfaces to interactive theorem provers (Isabelle [21] and Coq [22]) (see Figure 1). The interactive theorem prover interfaces make it possible for Jahob to, when necessary, leverage human insight to prove arbitrarily complex formulas requiring arbitrarily sophisticated reasoning.

The reason integrated reasoning is so appropriate for our program verification tasks is the diversity of the generated verification conditions. The vast majority of the formulas in these verification conditions are well within the reach of even quite simple decision procedures. But there are always a few complex formulas that require sophisticated techniques. Integrated reasoning makes it possible for Jahob to apply multiple reasoning systems as appropriate: simple, fast solvers for simple formulas, and arbitrarily sophisticated (and therefore arbitrarily unscalable) solvers for complex formulas, and, if absolutely necessary, interactive theorem proving.

In recent years the automated reasoning community has developed a range of very powerful reasoning systems. We have found integrated reasoning to be an effective way to productively apply the combined power of these reasoning systems to verify complex and important program correctness properties. One particularly important aspect of integrated reasoning is its ability to leverage arbitrarily specialized reasoning systems that are designed to operate within arbitrarily narrow domains. The tight focus of these reasoning systems makes them irrelevant for the vast majority of the reasoning steps that Jahob must perform. But they are critical in enabling Jahob to deal effectively with the full range of properties that arise in the verification of sophisticated linked data structure implementations (and, we expect, will arise in other ambitious program verification efforts). Jahob's integrated reasoning technique makes it possible for these kinds of reasoning systems to make a productive contribution within a larger program analysis and verification system.

1.2 Proof Choice Point Selection

In our experience, automated reasoning systems encounter difficulties proving verification conditions when the corresponding proof search spaces have key choice points (such as quantifier instantiations, case splits, selection of induction hypotheses, and lemma decompositions) that the reasoning systems are unable to resolve in any relevant amount of time.

We have therefore developed a proof language that enables developers to resolve these key choice points [9]. Because this proof language is integrated into the underlying imperative programming language (Java), it enables developers to stay within a familiar conceptual framework as they work with choice points to shape the overall structure of the path through the proof search space. The automated reasoning systems in Jahob are then able to leverage this high-level guidance to perform all of the remaining proof steps. This approach appropriately leverages the complementary capabilities of the developers and the automated reasoning systems. Developers usually have the insight required to effectively shape the high-level structure of the correctness proof; automated reasoning systems excel at the detailed symbolic manipulation required to leverage the guidance to complete the proof.

Given the substantial reasoning capabilities of current reasoning systems and the rate at which these reasoning capabilities are improving, we expect technologies (such as the Jahob integrated proof language) that are designed to leverage high-level developer insight into the proof process to become increasingly viable. And because these techniques can significantly reduce the effort required to obtain a given result, we expect them to be increasingly deployed across a wide range of automated reasoning scenarios.

1.3 Program Survival

Data structure consistency provides, by itself, no guarantee that the program as a whole will execute correctly. Indeed, given the difficulty of obtaining full program correctness specifications (and, more fundamentally, the fact that no one can state precisely what most large software systems should do), we expect full program verification to remain beyond reach for the foreseeable future.

But it is possible, however, to systematically identify all of the errors that could cause a program to fail outright. And because standard techniques such as testing can deliver programs that operate reliably on almost all inputs, the elimination of fatal errors is a key step towards obtaining software systems that can survive the inevitable remaining errors to continue to deliver acceptable service.

We have developed a set of (in most cases quite simple) dynamic techniques that, together, enable programs to survive virtually all errors [1–7]. The experimental results show that these techniques are surprisingly effective in enabling programs to survive otherwise fatal errors so that they can continue to provide effective service to users.

1.4 Static Verification of Survival Properties

In this paper we explore the possibility of using program analysis and verification technology to statically guarantee the absence of fatal errors in software systems. A recurring theme is that relatively simple and scalable techniques that can operate effectively across large parts of the program should be sufficient to verify the absence of fatal errors in almost all of the program.

But there will always be (for reasons of efficiency and desired functionality) complex parts of the program that are difficult if not impossible to analyze using these techniques. For these parts of the program we advocate the use of sophisticated heavyweight program analysis and verification techniques that potentially involve the developer, in some cases quite intimately, in the verification process. We see two keys to the success of this approach:

- **Modularity:** It is practical to apply heavyweight reasoning techniques only to a feasibly sized region of the program. Modularity mechanisms (such as abstract data types) that restrict the regions of the program to which these techniques must be applied are essential to the success of this approach.
- **Developer Interaction:** We have found both integrated reasoning and proof choice point selection to play critical roles in enabling successful data structure consistency proofs. Given the quite sophisticated properties that will inevitably arise, we believe that both of these techniques will also prove to be critical to the verification of other survival properties.

For any particular program correctness property, there is a temptation to develop specification languages and verification techniques that are tailored to that property. We have chosen, in contrast, to use a general purpose specification and verification approach. The disadvantage of this general approach is that for any specific property it can require more developer specification and verification effort than an approach tailored for that property. But the range of properties that our general approach can address justifies the engineering effort required to build a powerful and versatile reasoning system that the developer can use to specify and verify virtually any desired property. In particular, the system can usually smoothly extend to handle unforeseen extensions of the original properties. Tailored approaches, in contrast, usually have difficulty supporting unforeseen extensions, in which case the developer is left with no good specification and verification alternative at all.

We focus on the static verification of several properties that are especially relevant to program survival: data structure consistency, the absence of memory errors such as null pointer dereferences and out of bounds accesses, the absence of memory leaks, and control flow properties such as the absence of infinite loops. In all of these cases, with the exception of memory leaks, we expect a dual analysis approach to be effective in making it possible to statically verify the desired property. In particular, we expect simple and scalable techniques to suffice for the vast majority of the program, with standard encapsulation mechanisms such as abstract data types providing the modularity properties required to apply heavyweight techniques successfully to those feasibly small regions of the program that require such techniques. While the current focus is on verifying complex properties in targeted parts of the program, in the longer run we expect the construction of effective scalable analyses for the simpler properties that must hold across large regions of the program to prove to be the more challenging problem.

2 Dynamic Techniques

Before considering static verification techniques for survival properties, we first review our experience with a collection of dynamic survival techniques. We obtained these techniques by analyzing the different ways that programs could fail, then developing a simple technique that enables the program to survive each class of failures.

In comparison with the static techniques considered in this paper, the great advantage of these dynamic techniques is their simplicity — instead of relying on potentially quite heavyweight program analysis and verification technology, the techniques simply change the semantics of the underlying model of computation to completely eliminate the targeted class of failures. The insight here is that standard models of computation are unforgiving and brittle — the unstated assumption is that because programs should be perfect, the developer should bear all of the responsibility for making the program execute successfully. Our new models of computation, in contrast, acknowledge that although developers make mistakes, these mistakes need not be fatal in the context of a more forgiving model of computation that works harder to make the program succeed. Our experimental results show that the resulting nonstandard models of computation can be quite effective in practice in enabling programs to execute effectively through situations that would cause them to fail with standard brittle models of computation.

2.1 Failure-Oblivious Computing

Programs that use failure-oblivious computing check each memory access (either statically or dynamically) for memory errors. On write memory errors (for example, an out of bounds write or write via a null pointer), the program simply discards the write. On read memory errors, the program simply makes up a value to return as the result of the read [4]. In effect, this technique changes the semantics of the underlying programming language to make *every* memory access that the program can ever execute succeed. Experimental results indicate that, in practice, this technique enables programs to successfully survive otherwise fatal memory errors. And it provides an absolute guarantee that a memory error will never terminate the execution. Discarding out of bounds writes also helps eliminate data structure corruption errors that would otherwise occur on out of bounds writes. The standard way to apply failure-oblivious computing is to use a compiler to introduce the memory error checks (although it is, in principle, possible to apply the technique at the binary or even hardware level [23]).

Note that throwing exceptions on memory errors in the hope of invoking a developer-provided exception handler that can recover from the error is usually totally ineffective. In practice, the exception usually propagates up to the top-level exception handler, which terminates the program. And in general, it can be difficult for developers to provide effective exception handlers for unanticipated exceptions. Most memory errors are, of course, unanticipated — after all, if the developer had thought a memory error could occur, he or she would have written different code in the first place.

2.2 Loop Termination

Infinite loops threaten the survival of the program because they deny parts of the program access to a resource (the program counter) that they need to execute successfully. The following bounded loop technique completely eliminates infinite loops [24]: [1]

- **Training:** Execute the program for several training runs. For each loop, record the *observed maximum* — i.e., the maximum number of iterations the loop performed during any training run.

[1] Of course, any program that is designed to execute for an unbounded period of time must have at least one unbounded loop (or other form of unbounded execution such as unbounded recursion). For example, many event-driven programs have an event-processing loop that waits for an event to come in, processes the event, then returns back to the top of the loop to wait for the next event. It is relatively straightforward for the developer to identify these loops and for the system to not apply the infinite loop termination technique to these loops.

- **Iteration Bound:** Select a *slack factor* (typically a number around several thousand) and, for each loop, compute an *iteration bound* — i.e., the slack factor times the observed maximum for that loop. If the loop never executed during the training runs, simply use the maximum iteration bound for loops that did execute during training runs.
- **Iteration Bound Enforcement:** If necessary, transform the program to add an explicit loop counter to each loop. During production runs, use the loop counter to exit the loop whenever the number of iterations exceeds the iteration bound. If the loop exits after executing more iterations than the observed maximum number of iterations from the training runs but fewer iterations than the iteration bound, update the iteration bound to be the slack factor times the number of iterations from the current execution of the loop.

2.3 Cyclic Memory Allocation

Like infinite loops, memory leaks are a form of unbounded resource consumption that enable one component to exhaust resources required for the program to survive. It is possible to completely eliminate memory leaks by allocating a fixed size buffer for each memory allocation site, then allocating objects cyclically out of that buffer [24]. Specifically, the nth object allocated at each site is allocated in slot $n \bmod s$, where s is the number of objects in the buffer. If the program only accesses at most the last s objects allocated at the corresponding site, this transformation does not affect the correctness of the program.

The potential disadvantage of cyclic memory allocation is that it may allocate multiple live objects in the same slot. The result is that writes to one object will overwrite other objects allocated in that same slot. Our experimental results indicate that overlaying live objects in this way typically causes the program to gracefully degrade rather than fail [24]. We attribute this property, in part, to the fact that the transformation (typically) preserves the type safety of the program.

2.4 Data Structure Repair

In the presence of failure-oblivious computing corrupted data structures are, by themselves, incapable of causing an outright program failure. They can, however, significantly disrupt the execution of the program and make it difficult for the program to execute acceptably. Data structure repair, which is designed to restore important consistency properties to damaged data structures, can significantly improve the ability of the program to deliver acceptable results [1, 6, 7]. We have found that this result holds even when the repair is unable to completely reconstruct information originally stored in the data structure (typically because the data structure corruption error destroyed the information before the repair was invoked).

3 Data Structure Consistency

Our experience with data structure repair and failure-oblivious computing shows that data structure consistency can be a critical component of meaningful program survival. We propose a multistage approach to the static verification of data structure consistency. The first stage is to obtain fully verified implementations of standard abstract data types. This goal has been largely achieved in Jahob program verification system [9, 8, 10]. The second stage is to verify properties that involve multiple abstract data types. Building on verified abstract data type implementations, the Hob project made significant progress towards this goal [25–28], but challenges still remain.

3.1 Jahob

Jahob is a general program verification system with a powerful specification language based on higher-order logic. Because the specifications take the form of specialized comments, it is possible to use standard Java implementation frameworks to execute specified programs. Given these specifications, the Jahob implementation processes the program to generate verification conditions. It then uses its underlying integrated reasoning system to prove these verification conditions.

3.2 The Jahob Specification Language

Jahob specifications use primitive types (such as integers and booleans), sets, and relations to characterize the abstract state of the data structure. A verified abstraction function establishes the correspondence between the concrete values that exist when the program executes (the implementation directly manipulates these values) and the abstract state in the specification (which exists only for verification purposes). Method contracts, class invariants, and annotations within method bodies use classical higher-order logic to express the desired properties of the data structure interface and implementation.

Specification Variables In addition to concrete Java variables, Jahob supports *specification variables*, which do not exist during program execution but are useful to specify the behavior of methods without revealing the underlying data structure representation. In addition to other purposes, developers use specification variables to identify the abstract state of data structure implementations. Abstraction functions specify the connection between the concrete Java variables and the corresponding specification variables.

Method Contracts A method contract in Jahob contains three parts: 1) a precondition, written as a requires clause, stating the properties of the program state and parameter values that must hold before a method is invoked; 2) a frame condition, written as a modifies clause, listing the components of the state that the method may modify (the remaining components remain unchanged); and 3) a postcondition, written as an ensures clause, describing the state at the end of the method (possibly defined relative to the parameters and state at the entry of the method). Jahob uses method contracts for assume/guarantee reasoning in the standard way. When analyzing a method m, Jahob assumes m's precondition and checks that m satisfies its postcondition and the frame condition. Dually, when analyzing a call to m, Jahob checks that the precondition of m holds, assumes that the values of state components from the frame condition of m change subject only to the postcondition of m, and that state components not in the frame condition of m remain unchanged. Public methods omit changes to the private state of their enclosing class and instead use public specification variables to describe how they change the state.

Class Invariants A *class invariant* can be thought of as a boolean-valued specification variable that Jahob implicitly conjoins with the preconditions and postconditions of public methods. The developer can declare an invariant as private or public (the default annotation is private). Typically, a class invariant is private and is visible only inside the implementation of the class. Jahob conjoins the private class invariants of a class C to the preconditions and postconditions of methods declared in C. To ensure soundness in the presence of callbacks, Jahob also conjoins each private class invariant of class C to each reentrant call to a method m declared in a different class D. This policy ensures that the invariant C will hold if $D.m$ (either directly or indirectly) invokes a method in C. To make an invariant F with label l hold less often than given by this policy, the developer can write F as $b \rightarrow I$ for some specification variable b. To make F hold more often, the developer can use assertions with the shorthand ($\mathsf{theinv}\, l$) that expand into F.

Loop Invariants The developer states a loop invariant of a while loop immediately after the while keyword using the keyword invariant (or inv for short). Each loop invariant must hold before the loop condition and be preserved by each iteration of the loop. The developer can omit conditions that depend only on variables not modified in the loop — Jahob uses a simple syntactic analysis to conclude that the loop preserves such conditions.

3.3 The Jahob Integrated Proof Language

Conceptually, most reasoning systems search a proof space — they start with a set of known facts and axioms, then (at a high level) search the resulting proof space in an attempt to discover a proof of the desired consequent fact. We have found that, in practice, when automated reasoning systems fail, they fail because there are key choice points in the proof search space that are difficult for them to resolve successfully. We have therefore developed a proof language, integrated into the underlying imperative programming language, that enables the developer to resolve such choice points [9]. Examples of such choice points include lemma decompositions, case splits, universal quantifier instantiations, and witness identification for existentially quantified facts. We have also augmented the language with constructs that allow developers to structure proofs by contradiction and induction.

Although we focus on the constructs that developers use to resolve choice points, Jahob provides a full range of proof constructs. Developers can therefore provide as much or as little guidance as desired. It is even possible for a developer to use the integrated proof language to explicitly perform every step of the proof.

Finally, Jahob provides a construct that enables developers to deal with a pragmatic problem that arises with modern reasoning systems. In practice, these reasoning systems are very sensitive to the *assumption base* — the assumptions from which to prove the desired consequent fact. If the assumption base is too large or contains too many irrelevant assumptions, the search space becomes too difficult for the reasoning system to search effectively and it fails to find a proof. Jahob therefore allows developers to name and identify a set of assumptions for the reasoning systems to use when attempting to prove a specific fact. The resulting precise identification of a minimal assumption base can often enable the reasoning systems to prove desired facts without any additional assistance. This capability can be especially important when verifying complex data structures (because the reasoning systems place many properties into the assumption base during the course of the verification).

3.4 Verification Condition Generation and Integrated Reasoning

Jahob produces verification conditions by simplifying the Java code and transforming it into extended guarded commands, then desugaring extended guarded commands into simple guarded commands, and finally generating verification conditions from simple guarded commands in a standard way [10, 8]. It then splits the verification conditions into subformulas and, with the aid of formula approximation, uses the integrated automated reasoning systems to prove the subformulas [10, 8]. It runs each automated reasoning system with a timeout and, on multicore machines, supports the invocation of multiple automated reasoning systems in parallel to prove a given subformula. Our current system verifies most of our data structure implementations within several minutes [8]. Our binary search tree implementation, with a verification time of an hour and forty-five minutes (primarily because of time spent in the MONA decision procedure), is an outlier.

3.5 Hob

Certain data structure consistency properties involve multiple data structures. The goal of the Hob project was to develop techniques for statically verifying such properties. Hob worked with set abstractions — it modelled the state of each data structure as an abstract set of objects. Developers could then use a standard set algebra (involving the boolean operators along with operations such as set inclusion and set difference) to state the desired consistency properties [25–28].

Although Hob contained techniques for verifying the consistency of individual data structure implementations, in the long run a more effective approach would be to first use a system like Jahob to verify the full functional correctness of individual data structure implementations, then use a more scalable system like Hob to verify properties involving multiple data structures. The Hob verification would, of course, work with the interfaces for the individual data structures (these interfaces would be verified by Jahob or a system like Jahob), not the data structure implementations. This approach would appropriately leverage the relative strengths of the two systems.

Our experience with Hob indicates that sets of objects provide a particularly compelling abstraction. In many domains it is possible to model user-level concepts with sets of objects; in these domains the resulting Hob specifications often capture properties that are directly relevant to the user. This relevance stands in stark contrast to standard specifications, which tend to focus heavily on internal implementation details as opposed to high-level concepts that are directly meaningful to users.

Extending Hob to support relations would significantly increase its expressiveness and utility. This extension would enable Hob to support more sophisticated consistency constraints — for example, that one data structure contains every object in the domain of a map implemented by another data structure. The challenge with this extension is developing the scalable analyses required to verify these more expressive constraints.

3.6 Self-Defending Data Structures

Many verification approaches (including Jahob and Hob) use assume/guarantee reasoning. Each procedure has a precondition that it assumes is true upon entry. It is the responsibility of the client to ensure that the precondition holds. Each procedure also has a postcondition, which the procedure implementation guarantees to be true assuming that the precondition holds upon entry to the procedure. Encapsulated invariants capture the key consistency properties.

With this approach, the data structure consistency depends both on the implementation and on the client — if the client fails to satisfy a precondition, there is no guarantee that the implementation will preserve any encapsulated invariants. Moreover, the standard preconditions for many abstract data types are often quite complicated and difficult to verify with the kind of scalable analyses required to successfully verify that clients correctly satisfy the preconditions.

We therefore advocate the use of *self-defending* data structures with empty preconditions (such data structures can, of course, have encapsulated invariants). Self-defending data structures contain all of the checks required to ensure that they remain consistent regardless of client behavior. The advantage is that such data structures completely eliminate the need for client analyses in the verification of the consistency of individual data structures.

4 Infinite Loops

In the last several years researchers have developed a set of techniques for statically verifying that loops terminate [29]. We expect that, in general, researchers will be able to develop techniques that are effective in proving that the vast majority of loops terminate. But we also expect certain kinds of loops to continue to be beyond the reach of any practically deployable lightweight static analysis. Two examples of such loops are loops that operate on linked data structures (the termination of these loops may depend on complex data structure consistency properties) and loops whose termination depends on the convergence of complex numerical algorithms.

Given the tractable size of abstract data types, we advocate the use of heavyweight formal reasoning techniques (much like those that the Jahob system supports) to prove termination properties. Because abstract data types are standard, widely used components, they can justify large formal reasoning efforts.

We anticipate that a similar approach would work for many numerical algorithms. But formal reasoning techniques for numerical algorithms are less developed than those for more discrete or symbolic algorithms. And in many cases properties such as termination depend on subtle mathematical properties of discretized representations of continuous quantities. It is unclear the extent to which formally verified termination proofs will become practical for widespread use. One potential solution is simply to place a predetermined bound on the number of iterations of each loop (in a manner similar to the loop termination technique described above in Section 2.2).

5 General Control Flow Anomalies

Infinite loops are a special case of more general control flow anomalies that can prevent the program from executing components required to provide acceptable service. For example, many event-driven programs have an event processing loop. In many cases it may be worthwhile to develop static analyses that reason about the control flow to prove that every execution will, in a finite amount of time, return back to the top of the event processing loop. We anticipate that the major complication (to the extent that there is one) will be reasoning about the behavior of the program in the face of exceptions and explicit program exits. Developer-provided assertions and safety checks can be especially counterproductive in this context. Developers are often overly conservative about the conditions they check — in our experience, programs are often able to survive violations of checks that developers have added to terminate the program in the face of unexpected executing conditions or state changes. One straightforward approach is to simply excise all calls to primitives that terminate the program [3].

6 Memory Errors

Reasonably simple and usable augmented type systems exist that are capable of statically guaranteeing the absence of null pointer dereferences [30]. Statically verifying the absence of out of bounds array accesses is a much more challenging and complex task, in large part because of the need to reason about (potentially complex) array indexing expressions [31]. Once again, we expect the most productive approach to involve isolating complex array indexing computations inside abstract data types or similar modules, then using potentially heavyweight sophisticated reasoning techniques to prove the absence of array bounds violations. With this approach, simpler and more scalable techniques should be able to verify the absence of array bounds violations in the remaining parts of the code.

7 Memory Leaks

Statically verifying the absence of memory leaks given standard program semantics is, with current program analysis and verification technology, the most challenging task we consider in this paper. Techniques exist for finding certain classes of leaks in array-based data structure implementations [32]. Modified escape analyses should also be able to find leaks that occur when the program takes certain exceptional execution paths [33]. A key difficulty is that objects may remain reachable in linked data structures with the data structure interface enabling invocation sequences that can cause the computation to access the objects. But in some cases, because of restricted usage patterns in the data structure clients, these invocation sequences can never actually occur in the program as a whole. We are aware of no static analysis or verification techniques that are designed to operate successfully in this scenario. The difficulty of developing such techniques would depend heavily on the characteristics of the usage patterns that occur in practice.

8 Conclusion

Program survival is a key consideration in a world in which full program verification is unrealistic. Statically verifying key survival properties will involve scalable static analyses that operate over the vast majority of the program working in combination with sophisticated program verification technologies that leverage both heavyweight automated reasoning techniques and developer intervention to prove complex survival properties in targeted regions of the program. These technologies promise to significantly enhance our ability to deliver software systems that can successfully execute through errors to provide acceptable service to users.

References

1. Demsky, B., Rinard, M.: Data structure repair using goal-directed reasoning. In: Proceedings of the 2005 International Conference on Software Engineering. (2005)

2. Rinard, M.: Acceptability-oriented computing. In: 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion) Onwards! Session. (October 2003)
3. Rinard, M., Cadar, C., Nguyen, H.H.: Exploring the acceptability envelope. In: 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '05 Companion) Onwards! Session. (October 2005)
4. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., William S. Beebee, J.: Enhancing server availability and security through failure-oblivious computing. In: Proceeding of 6th Symposium on Operating System Design and Implementation (OSDI 2004). (2004)
5. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T.: A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In: Proceedings of the 2004 Annual Computer Security Applications Conference . (2004)
6. Demsky, B., Rinard, M.: Automatic detection and repair of errors in data structures. In: Proc. 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2003)
7. Demsky, B., Rinard, M.: Static specification analysis for termination of specification-based data structure repair. In: IEEE International Symposium on Software Reliability. (2003)
8. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008) . (June 2008)
9. Zee, K., Kuncak, V., Rinard, M.: An integrated proof language for imperative programs. In: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009) . (June 2009)
10. Kuncak, V.: Modular Data Structure Verification. PhD thesis, EECS Department, Massachusetts Institute of Technology (February 2007)
11. Weidenbach, C.: Combining superposition, sorts and splitting. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume II. Elsevier Science (2001) 1965–2013
12. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications **15**(2/3) (2002) 111–126
13. Barrett, C., Tinelli, C.: CVC3. In Damm, W., Hermanns, H., eds.: Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). Volume 4590 of Lecture Notes in Computer Science., Springer-Verlag (July 2007) 298–302 Berlin, Germany.
14. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: CADE. (2007)
15. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In Ramakrishnan, C.R., Rehof, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337–340
16. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: CADE. (2007)
17. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019. (1995)

18. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa (2006) Available at `www.SMT-LIB.org`.
19. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding Boolean Algebra with Presburger Arithmetic. J. of Automated Reasoning (2006) http://dx.doi.org/10.1007/s10817-006-9042-1.
20. Kuncak, V., Rinard, M.: Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In: CADE-21. (2007)
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer-Verlag (2002)
22. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
23. Witchel, E., Rhee, J., Asanović, K.: Mondrix: Memory isolation for linux using mondriaan memory protection. In: 20th ACM Symposium on Operating Systems Principles (SOSP-20). (2005)
24. Nguyen, H.H., Rinard, M.: Detecting and eliminating memory leaks using cyclic memory allocation. In: Proceedings of the 2007 International Symposium on Memory Management. (2007)
25. Lam, P., Kuncak, V., Rinard, M.: Cross-cutting techniques in program specification and analysis. In: 4th International Conference on Aspect-Oriented Software Development (AOSD'05). (2005)
26. Kuncak, V., Lam, P., Zee, K., Rinard, M.: Modular pluggable analyses for data structure consistency. IEEE Transactions on Software Engineering **32**(12) (December 2006)
27. Lam, P., Kuncak, V., Rinard, M.: Generalized typestate checking for data structure consistency. In: 6th Int. Conf. Verification, Model Checking and Abstract Interpretation. (2005)
28. Lam, P.: The Hob System for Verifying Software Design Properties. PhD thesis, Massachusetts Institute of Technology (February 2007)
29. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006) . (June 2006)
30. Papi, M.M., Ali, M., Jr., T.L.C., Perkins, J.H., Ernst, M.D.: Practical pluggable types for java. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, Seattle, WA (July 2008)
31. Rugina, R., Rinard, M.C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. ACM Trans. Program. Lang. Syst. **27**(2) (2005)
32. Shaham, R., Kolodner, E., Sagiv, S.: Automatic removal of array memory leaks in java. In: Proceedings of the 9th International Conference on Compiler Construction. (2000)
33. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: OOPSLA, Denver (November 1999)