Research Plan:
Precise and Scalable Analyses for Reliable Software

Viktor Kuncak
EPFL

October 1, 2007

1 Extended Summary of the Research Plan

Software developed today contains many errors. If not discovered, such errors can have disastrous consequences. Static analysis and verification tools have a great potential for detecting and helping eliminate many of these errors. This potential is currently not fully realized because existing tools either detect only a limited range of errors, or fail to scale to substantial pieces of software. To resolve this problem we have developed an approach for combining precise and scalable techniques in software analysis. Our goal is to make this approach applicable to real-world software.

In our approach, the developer specifies software interfaces in a common specification language. Our systems use these interfaces along with an automated decomposition procedure to combine multiple analyses, decision procedures, and theorem provers, leveraging the strengths of each individual technique. This approach enables the development of software components with verified interfaces, supports precise analysis of selected regions of code, and supports scalable analysis of substantial code bases. We have implemented this approach within two software analysis systems.

The Hob analysis system. The Hob analysis system [64, 69] verifies programs written in the Hob language, an imperative memory-safe language with dynamically allocated cells and statically instantiated parameterized modules. Module interfaces in Hob are expressed in set algebra. Hob uses symbolic shape analysis and a decision procedure for monadic second-order logic over trees to verify data structure implementations. It uses an analysis that propagates set expressions to verify data structure uses as well as relationships between data structures. By analyzing properties in applications such as liquid simulation and web server, each of which uses dynamically allocated data structures, Hob demonstrated that a combination of precise and scalable analyses is effective.

The Jahob analysis system. We have started developing the Jahob system [63] for analyzing Java programs, with the goal of transferring these ideas into real real-world programming languages and verifying a richer set of properties. Jahob allows expressing and verifying richer interfaces, including interfaces of dynamically instantiable data structures and numerical properties. To enable verification of richer interfaces, we developed a technique that can not only combine multiple analyses operating in different regions of code, but also combine multiple automated reasoners when proving each proof obligation. Using this technique, we incorporated into the system first-order theorem provers [18] and satisfiability modulo theories solvers. We have also developed and incorporated new decision procedure for sets with cardinality constraints [65, 66], enabling the verification of invariants about sizes of data structures.

Proposed directions. We propose to develop the Jahob analysis system along several directions. Our expectation is that these developments will make Jahob applicable to components of real-world software, and that they will advance the state of the art in static analysis, verification, decision procedures, and methodology for reliable software development.

- **Scalable high-level analyses.** We will develop analyses capable of propagating rich interfaces across substantial pieces of code. These analyses will correspond to the high-level analysis in the Hob system [69], but will support Jahob's more expressive specification language [63]. As the guideline in implementing scalable constraint solvers we will explore techniques from description logic systems [106], set constraint solvers [62], and systems based on binary decision diagrams [109]. To support rich invariants we will also consider new classes of constraints such as sets and multisets with symbolic cardinality bounds. We will apply high-level analyses to verify uses of container data structures, including iterators, to verify relationships between containers, and to detect errors such as null dereference, incorrect sequence of operations, and unreachable code.

- **Data structure analysis.** We will further improve the effectiveness of Jahob's data structure analysis. This analysis is one of the current strengths of Jahob, but the precision required to verify that a real-world data structure implements the desired interface can be comparable to full program verification and still presents a challenge for current systems. To address this challenge we will augment our current decision procedure based on monadic second-order logic with faster decision procedures, using recent results on logics for regular trees [45]. Furthermore, we will develop a library of transfer functions for common data structure manipulation patterns, with the goal of avoiding most of the decision procedure invocations during the analysis. With these improvements we expect to be able to verify data structures present in widely used collection libraries.

- **Methodology for modular verification.** To enable verification of detailed properties in large code bases, we will combine the high-level analysis with the data structure analysis. Such a combination requires a methodology for modular analysis that enforces encapsulation and is sound in the presence of mutation, aliasing, and callbacks. We plan to develop a generic mechanism for specifying modular analysis methodologies while ensuring soundness. Using this mechanism, we will explore new points in the design space of encapsulation disciplines and compare them to recently proposed approaches [10] in terms of programmer flexibility and the complexity of generated proof obligations.

- **Extending the applicability of the system.** We will develop several techniques to make Jahob applicable in multiple scenarios and by users with different expertise. We will organize system architecture into the front-ends, the intermediate language verifier, and the theorem proving engine; each of these components will be useful on its own. We make Jahob input suitable for verifying constraints from graphical modelling notation, enabling its use not only during code development but also during software design. We will extend Jahob's annotation language to allow users complete control over the proving process and thereby support verification of arbitrarily complicated properties. Finally, we will integrate Jahob with a run-time checker and a finite domain constraint solver [79], which will allow users to debug specifications and code using specification-based testing.

The proposed work will benefit from interactions with Professor Wei Ngan Chin from the National University of Singapore, Dr. Pierre Genevés from INRIA and CNRS, France, Dr. Rustan M. Leino from Microsoft Research Redmond, USA, Professor Darko Marinov from the University of Illinois Urbana-Champaign, USA, Professor Andreas Podelski from the Freiburg University, Germany, and Professor Martin Rinard from the Massachusetts Institute of Technology, USA. This work will also benefit from interactions with our colleages at EPFL, including Professor George Candea, Professor Thomas Henzinger, Professor Dejan Kostic, Professor Martin Odersky, Professor Alain Wegmann, and Professor Willy Zwaenepoel.

2 Background: The State of the Art

We next survey some of the relevant work in verification and static analysis and summarize its relationship to this proposal.

2.1 Program Verification

At the core of our proposals is a view of modern program verification as an integrating technology for a range of automated approaches that have been developed to address specific aspects of software reliability.

Evolution of program verification. Program verification has a long tradition and has resulted in tools such as the Program Verifier [60], Gypsy [47], Stanford Pascal Verifier [83], VDM [57], the B method [1], RAISE [30], and Larch [49]. This work established the principles of reducing verification to theorem proving. It also identified difficulties of the program verification problem, both in the automating the proofs of verification conditions and in user effort needed to devise appropriate specifications and loop invariants. More recent work on ESC/Modula-3 [36] and ESC/Java [42] reflects an important philosophical view shift from the full program verification ideal to building immediately usable tools that check common program errors using similar underlying techniques [83]. These ideas influence the scope of our proposal: we aim at building useful tool for the price of verifying potentially simpler properties and using a simplified (but well defined) programming language semantics.

Recent tools. Among the recently developed tools are ESC/Java2 [22] and Spec# [10], which share architecture similar to our Jahob system, but differ in loop invariant inference techniques and deployed theorem provers and decision procedures. To the best of our knowledge these tools are not integrated with shape analyses such as [112, 111] and use instead simpler techniques for loop invariance inference [41, 43]. In terms of theorem proving technology these systems use primarily satisfiability modulo theory solvers [92], whereas Jahob also uses resolution-based theorem provers [18], specialized decision procedures [111, 66], and interactive theorem provers [117]. One of the goals of our proposal is to collaborate with researchers developing these tools to exchange ideas and create a common set of benchmarks that will foster further cross-fertilization of techniques deployed in these systems.

Modular verification methodology. Data refinement [56, 33] is the general technique of soundly replacing a complex piece of software with a simpler one. However, there are difficulties in applying these techniques in modern object oriented programs, because the relationship between an object and its state is not given syntactically but depends on references in the heap [71]. Ownership types [19, 25] can approximate the relationship between the object and its auxiliary objects. Naumann

and Barnett [82] make explicit the binary ownership relation that is implicit in ownership types. One of the main features of Spec# [10] is a modular verification methodology that correctly handles such situations. The verification methodology in Jahob [63, Section 3.2.8] is more restrictive than one in Spec#. In the proposed work we will design a generic mechanism for exploring existing and new modular verification methodologies.

The role of interactive proof. Several tools support interactive theorem proving in the verification process, including LOOP [54], Krakatoa [38, 76], KIV [9], KeY [2], as well as the embeddings [98], [74]. In our experience [117] such combination is useful both in verifying very complex properties and in debugging the system itself [63, Section 4.2]. One of the goals of the present project is to make such verification available through program annotations that are close to standard programming constructs, in the hope of making it more accessible to software developers. The overall goal of the project, however, is to reduce the need for interaction as much as possible by advancing the automated methods.

2.2 Abstract Interpretation

One of the central problems in verification of safety properties is automated computation of reachability invariants, which can be formalized as abstract interpretation [29]. An analysis system can use such invariants to check that program satisfies desired specifications and avoids execution errors. Researchers have developed numerous analyses based on abstract interpretation. These analyses differ in classes of invariants that they can synthesize and analysis efficiency that determines their scalability. One of the goals of our system is to enable integration of a wide range of such analyses including scalable analyses proven to work on large code bases and shape analyses that can verify detailed data structure properties.

Scalable analyses. Many of the scalable program analyses have been developed in the context of pointer analysis [5, 101, 109, 16, 50] flow-sensitive typestate analysis [31], analysis of ML exceptions [3], and approximation of run-time types [4]. Many scalable analyses separate the problem of generating a set of constraints from code from the problem of solving the generated constraints using dedicated solvers. In the proposed work we will investigate the applicability of such existing and new constraint solving approaches to the problem of propagating properties specified by expressive preconditions and postconditions.

Data structure analyses. On the other side of the precision/efficiency spectrum of static analyses are shape analyses, which can analyze detailed properties of mutable data structures. Many previous shape analyses used specialized graph-like structures to represent abstract program state [96, 95, 46]. Jahob uses a new symbolic approach to shape analysis implemented in a component called Bohne [112, 111, 110, 90] and builds on the advances in predicate abstraction [8, 52] to automatically discover new predicates and refine the abstraction domain. In the context of the TVLA system [72, 73] researchers have also developed symbolic approaches [114, 115] as well as approaches for refining the abstraction domain [75] and inferring analysis transfer functions [93]. Separation logic approaches also use formulas to represent program state and have proven effective for verifying shape properties [70, 48, 15]. We plan to explore the potential of such techniques for improving the performance of Jahob's shape analysis. Jahob's data structure analysis proves not only preservation of shape invariants, but also invariants such as sorting and changes to data structure content. Such invariants can be handled by the system [84] that reasons about shape, sortedness, sizes, and data structure content. In the long term we expect to combine separation

logic reasoning with classical reasoning in Jahob.

2.3 Software Model Checking

Model checking techniques [27] automatically prove properties of finite state systems. They have proved successful in hardware verification where the state is typically finite [58]. Software systems have practically infinite state space, but abstraction techniques make model checking applicable in those cases as well. Two forms of abstraction are relevant for the research we are proposing.

Finitization. The first approach is to introduce bounds on parameters of the system such as the values of variables and the number of allocated objects, or to constrain code execution by bounding the number of loop iterations [103, 20, 59]. Such approaches usually do not guarantee the absence of errors (with the exception of specific classes of properties [7]). However, an error found using this approach usually indicates an error in the original infinite-state system. Such approaches are therefore very likely to be useful in the initial phases of verification when specification and the code contain many errors. We plan to deploy such approaches in the context of Jahob as a way of debugging specifications and code before attempting to prove the absence of errors.

Predicate abstraction. In recent years researchers developed counterexample-driven predicate abstraction [8, 51, 55, 52], a promising combination of model checking, abstract interpretation, and theorem proving. Jahob's shape analysis is based on generalization of these techniques to a richer, quantified, semantic domain [90], and benefits from the improvements in predicate abstraction techniques. With our collaborators we will continue developing symbolic shape analysis and explore the possibility of tuning the precision of abstract domain to obtain performance comparable to combinations of TVLA shape analysis with Blast predicate abstraction [17].

2.4 Expressive Static and Dynamic Type Systems

Type systems encourage program correctness by integrating reasoning into the programming language itself. Expressive type systems based on dependent types [113], refinement types [44] and recently developed Hoare Type Theory [81] enable developers to express proofs of detailed properties of programs. We propose to develop annotation systems that have similar benefits but are based on classical logic and imperative programming. Soft typing [21, 4] and hybrid type checking [40] combine static and dynamic type checking. We will explore such combination for Jahob specifications by developing a run-time checker that communicates with Jahob's static analyzers. Existing annotations languages that have both static and runtime checkers include Spec# [12, 11] and the Java Modelling Language [28, 23]. The Eiffel programming language [78] has a long tradition of run-time checking and is being extended with static verification as well. We hope to benefit from such related efforts elsewhere by exchanging ideas and verified artifacts using common formats for verification such as BoogiePL [34].

2.5 Theorem Proving and Constraint Solving

Automated theorem provers such as Simplify [35, 83] have long presented the core technology in program verifiers. Recent advances that incorporate insights from SAT solvers have led to substantial performance improvements [92, 13, 32] that, combined with hardware advances enable verification of more complex properties than before. First-order theorem provers have also made remarkable advances over past decades [99, 108, 107] and are successfully used in verification [18, 91].

We also found that for expressive properties we need to consider specialized decision procedures such as decision procedure for monadic second-order logic over trees [61]. In the proposed work we plan to integrate faster decision procedures that can express similar properties [45].

For debugging specifications and code we will explore the uses of finite constraint solvers such as Korat [20, 79] as well as the use of model finders [53, 105, 24]. We believe that these techniques will substantially increase the usability of Jahob.

Jahob uses the Isabelle interactive theorem prover [87] as a semantic foundation and a fall-back for proving verification conditions not proved by other techniques. Furthermore, interactive theorem provers are increasingly being integrated with decision procedures and automated provers [77, 100, 14] which makes their languages appealing as a notation for combining different reasoning techniques. The simple combination approach developed in Jahob [63, Section 4.4] could be useful as a technique for combining provers for expressive languages.

3 Prior and Current Research of the Applicant

We have contributed several results in the area of analysis algorithms and systems, decision procedures and verification case studies. We next describe our past and current work on building analysis systems and developing the underlying algorithms.

3.1 Hob System

To explore the idea of combining multiple analyses using verified set interfaces, we have developed the Hob analysis system [64, 111, 67, 117, 69]. Hob analyzes programs written in the Hob language, an imperative memory-safe language with dynamically allocated cells and statically instantiated parameterized modules. Each module in Hob consists of an implementation section, a specification section, and an abstraction section.

Figure 1 shows some of the operations in a Hob module that describes a doubly-linked list with an iterator. Note that the specification section abstracts the doubly-linked structure using an abstract set, `Content`. It also abstracts the state of the iterator using another set, `Iter`, which stores those elements of `Content` that still remain to be iterated over. These two sets provide a natural abstract description of a container with an iterator. Hob uses a shape analyses [64, 111] to verify that data structures such as linked list correctly implement their set specifications. In this process Hob shows that data structure operations avoid memory errors and perform the change to data structure content described by the `ensures` clause, with the meaning of sets specified by the abstraction function given in the `abst` section of the module. Because they refer to content change, such properties are stronger than properties analyzed by many other shape analyses [48].

Having verified that data structure implementations satisfy their set specifications, Hob then uses such specifications to verify the uses of data structures and relationships between data structures. For example, the following code fragment processes all elements of a the doubly-linked list with an iterator.

```
DLLIter.openIter();
while (!DLLIter.isLastIter()) {
  Node n = DLLIter.nextIter();
  process(n);
}
DLLIter.closeIter();
```

```
impl module DLLIter {
  format Node { next : Node; prev : Node; }
  var root, current : Node;

  proc isEmpty() returns e:bool {
     return root==null;
  }
  proc remove(n : Node) {
    if (n==current) {  current = current.next;  }
    if (n==root) { root = root.next; }
    Node prv = n.prev, nxt = n.next;
    if (prv!=null) { prv.next = nxt; }
    if (nxt!=null) { nxt.prev = prv; }
    n.next = null; n.prev = null;
  }
  proc openIter() { current = root; }
  proc nextIter() returns n : Node {
    Node n1 = current;
    current = current.next;
    return n1;
  }
  proc isLastIter() returns e: bool {
    return current==null;
  }
}

abst module DLLIter {
  use plugin "PALE";
  ...
  Content = { n : Node | "root<next*>n" };
  Iter = { n : Node | "current<next*>n" };
}
```

```
spec module DLLIter {
  format Node;
  specvar Content, Iter : Node set;
  invariant Iter in Content;

  proc isEmpty() returns e:bool
    ensures e' <=> (card(Content') = 0);
  proc remove(n : Node)
    requires card(n)=1 & (n in Content)
    modifies Content, Iter
    ensures (Content' = Content - n) &
            (Iter' = Iter - n);

  proc openIter()
    requires card(Iter) = 0
    modifies Iter
    ensures (Iter' = Content);
  proc nextIter() returns n : Node
    requires card(Iter)>=1
    modifies Iter
    ensures card(n')=1 & (n' in Iter) &
            (Iter' = Iter - n');
  proc isLastIter() returns e:bool
    ensures not e <=> (card(Iter') >= 1);
}
```

Figure 1: Hob Module for a Doubly Linked List with an Iterator

Hob can use set specifications alone to verify that preconditions of list operations are satisfied and that all list elements are processed at the end of the iteration. It can similarly verify disjointness of contents of multiple data structures even in the presence of operations that move elements between the data structures.

We have used Hob to verify applications such as a minesweeper game implementation, a water particle simulation, and a web server [67], each of which uses uses dynamically allocated data structures. In this verification effort we have combined shape analysis [111], interactive theorem proving [117], and analysis based on set algebra expressions [69], demonstrating that a combination of precise and scalable analyses is effective. We made contributions to the analysis of data structures [111], the analysis of data structure clients [69] and explored modularity problems [68].

3.2 Jahob System

The experience with the Hob system has led the applicant to start developing the Jahob system [63], which is the starting point of the research proposed in this application. Jahob shares with Hob several ideas: combining multiple techniques in the analysis of a given piece of software, using a common specification language as a way of combining these techniques, and using specification

variables to soundly abstract module details. Jahob aims to push these ideas further and explore them in a more realistic context: Jahob's implementation language is a subset of Java, and Jahob's specification language is a subset of Isabelle's higher-order logic [86]. We next outline the design of Jahob through an example and then discuss our current work on addressing the challenges of verifying expressive specifications in Jahob.

3.2.1 Jahob through an Example

We next consider the verification of detailed properties of a global singly-linked list in Jahob (for more details, please see [63, Chapter 2]). Figure 2 shows an example verification session. The upper section of the editor window shows the `List` class with the `addNew` method and Jahob annotations written inside special comments. Class annotations in this example introduce two specification variables of type set of objects: `nodes`, containing objects reachable from `root` along the `next` field, and `content`, containing objects stored in these nodes using the `data` field. The meaning of these specification variables is reflected in class invariants `nodesDef` and `contentDef`. Method contracts such as `addNew` uses specification variables to describe method behavior while keeping private the class representation in terms of `next` and `root` fields. Furthermore, class invariants, such as `sizeInv`, use specification variables to express consistency properties of the data structure. The body of `addNew` also contains assignments to specification variables, as well as a `noteThat` assertion that acts as a verified lemma that helps Jahob in the proof (in this case by indicating the assumptions needed to prove `sizeInv`).

The lower window section in Figure 2 shows the command used to invoke Jahob to verify `addNew`, as well as Jahob's output indicating a successful verification. A Jahob invocation specifies the name of the source file `List.java`, the method `addNew` to be verified, and a list of three *provers*: `spass`, `mona`, and `bapa`, used to establish proof obligations during verification. Jahob's output indicates a successful proof and shows that all three provers took part in the verification. We next describe how Jahob soundly combines different provers.

3.2.2 Proving Complex Formulas by Combining Multiple Provers

Jahob reduces the verification problem to a sequence of validity queries for formulas in higher-order logic. To prove such formulas Jahob first decomposes them into a conjunction of polynomially many simpler formulas, then attempts to apply each of the given provers to each of the conjuncts [63, Section 4.4]. Each prover typically accepts only a subset of higher-order logic formulas. Jahob therefore does prover-specific preprocessing of such conjuncts, which includes substituting certain equations in assumptions, rewriting constructs such as set comprehensions and set operations, and approximating constructs not supported by the prover. The result is a stronger proof obligation that is accepted by the language of the prover. The overall proof obligation succeeds if for each conjunct at least one prover succeeds in proving its approximation. The developers can increase the effectiveness of this simple approximation technique by introducing `noteThat` statements. A `noteThat` statement can only strengthen the proof obligation so its use is always sound. It transforms the proof obligation by effectively introducing a lemma (cut) into the proof that is specific to a given program point. It can optionally restrict the set of hypothesis in the conjunct to those whose name matches the given one.

We next describe the provers that we incorporated into Jahob using this technique.

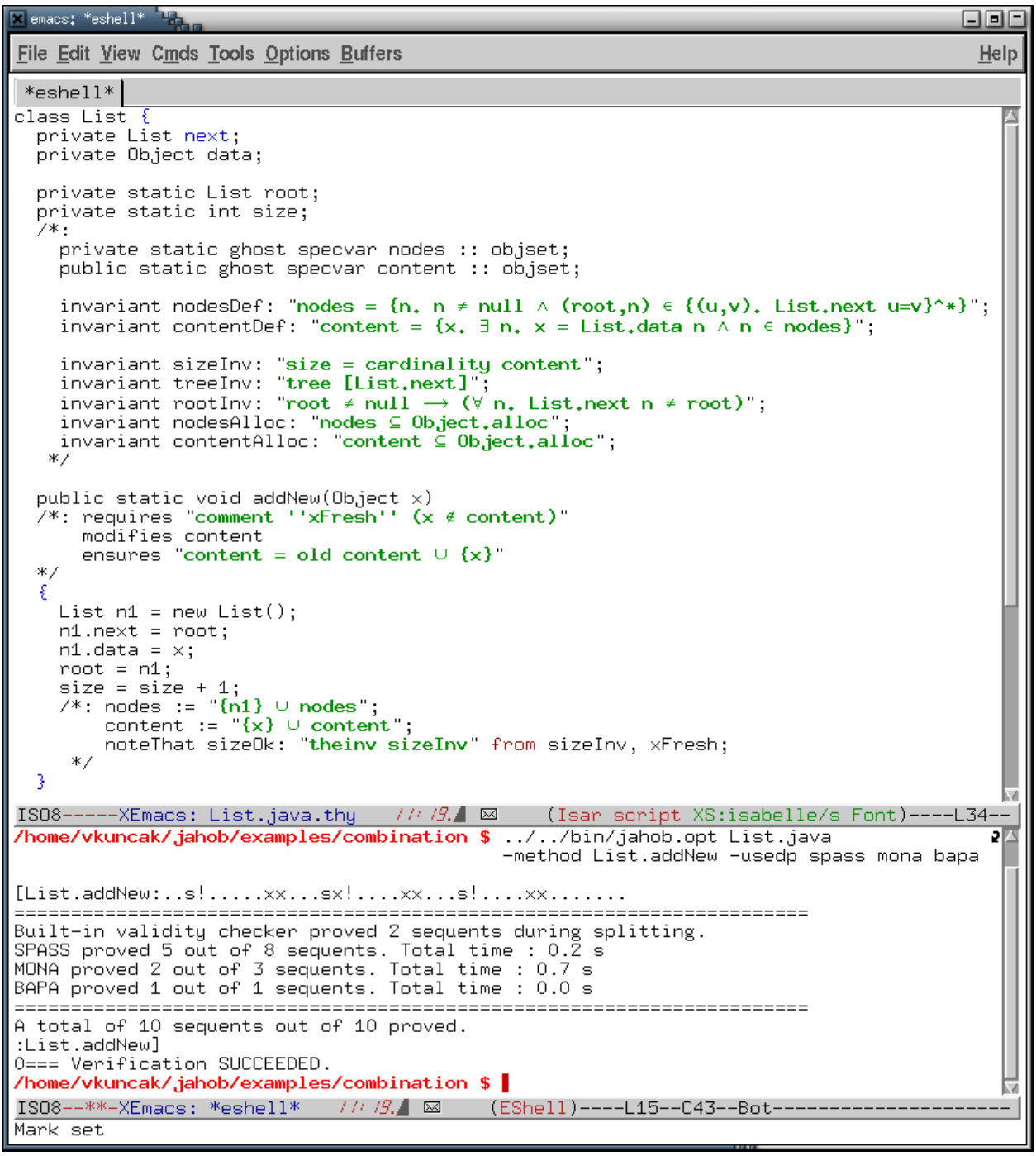

Figure 2: Screen shot of verifying a linked list implementation in Jahob

3.2.3 Leveraging First-Order Provers

Using our combination technique we were able to incorporate resolution-based theorem provers into Jahob, which is attractive because of their continuous improvement [108, 107, 99, 102]. Using

resolution-based provers we were able to verify data structures such as a trees, as well as a hash table implementation with operations insert, update, remove and rehash [18]. We have verified key data structure invariants, as well as postconditions such as

$$\mathsf{content} = (\mathsf{old\ content} \setminus \{(x, y).\ x = \mathsf{key}\}) \cup \{(\mathsf{key}, \mathsf{new\_value})\}$$

that precisely describe the change of the content relation after executing the hash table operation update(key, new_value). In the example from Figure 2, the translation into first-order logic and the use of SPASS [108] theorem prover successfully establishes 5 proof obligations, including the preservation of the contentDef invariant.

Using similar techniques, we were able to leverage satisfiability modulo theory solvers using the SMT-LIB standard interface [92, 13, 32]. We found SMT provers to be similarly useful, typically stronger with arithmetic and weaker with quantifier reasoning compared to resolution-based provers.

3.2.4 Leveraging Decision Procedures for Trees

We were also able to incorporate a decision procedure based on monadic second-order logic into our system. In the example from Figure 2 this technique proves the preservation of nodesDef and tree[Node.next] invariants.

Our translation expects formulas in the form of an implication where one of the assumptions is tree$[f_1, \ldots, f_n]$, meaning that graph given by fields $f_1, \ldots, f_n$ is a tree. It then approximates the entire formula by projecting it onto the tree structure with edges $f_1, \ldots, f_n$. To make this approximation more precise, we have developed a new techniques, field constraint analysis [111] that recognizes formulas of the form $\forall x, y.\ g(x) = y \rightarrow H(x, y)$ for a non-tree edge g, and uses them to soundly approximate the occurrences of g with formula H.

This technique enables us to precise analyze data structures with back pointers as well as data structures such as two-level skip list where H does not specify a function, overcoming limitations of previous tools [80]. Compared to approaches based on first-order theorem provers this approach enables the use of transitive closure, yielding explicit definitions for specification variables and often avoiding manual specification variable updates. More recently, as part of a student project in a class taught by the applicant, this approach was applied to verify not only acyclic but also to cyclic data structures, without any additional extensions to the underlying mechanism.

3.2.5 Developing and Using new Decision Procedures

Identifying relevant classes of specifications and developing decision procedures for them are among the the most significant developments in the context of Jahob. One class of decision procedures arises from verification of invariants such as sizeInv in Figure 2. The preservation of this invariant yields the verification condition of the form

$$x \notin \mathsf{content} \wedge \mathsf{size} = \mathsf{cardinality}(\mathsf{content}) \ \rightarrow\ \mathsf{size} + 1 = \mathsf{cardinality}(\{x\} \cup \mathsf{content})$$

This constraint belongs to a natural class that we call Boolean Algebra with Presburger Arithmetic because it contains both Boolean Algebra of sets and Presburger Arithmetic on the cardinalities of sets. This language admits quantifier elimination [37] so it is decidable, but no complexity results for the decision procedure were previously known. In [65] we describe an implementation of a decision procedure for quantified constraints and characterize its complexity. Subsequently we examined

quantifier-free constraints, for which all previous algorithms were running in non-deterministic exponential time. We found a new algorithm that runs in non-deterministic polynomial time [66], obtaining an exponential improvement. This improvement allows us to find counterexamples for certain invalid verification conditions formulas that we were previously unable to handle. Most recently, we have started extending these results to constraints that involve not only sets but also multisets in which elements can occur multiple times [89]. We have obtained new decision procedures and complexity bounds for quantifier-free constraints, and established undecidability of quantified constraints.

3.2.6 Runtime Checking

There are many errors in code that can be detected by a moderate amount of testing. Testing can similarly detect errors in specifications if the verification system has a facility to execute specifications at run time. We have recently started exploring issues involved in building such a system by using a runtime checking prototype built as an interpreter of Jahob's intermediate language [116]. One of the challenges in building a runtime checker for a program verification system is that the language of invariants and assertions is designed for simplicity of semantics and tractability of proofs, and not for run-time checking. Some of the more challenging constructs include existential and universal quantification, set comprehension, specification variables, and formulas that refer to past program states. We have examined the uses of these constructs that arise in specifications that we encountered and developed approaches for handling them.

We have also started exploring the problems of designing a specification language to facilitate both runtime checking and static verification. In addition to classical logic, we have explored runtime checking approaches for a subclass of separation logic specifications [85]. Finally, we are working on extending the functionality of constraint solvers for imperative predicates [79] to make them applicable to specification-based testing in verification tools such as Jahob.

4 Proposed Research Directions

Jahob is a promising verification system, but requires important further development to become applicable to real-world software. These developments must simultaneously happen at multiple fronts and require substantial algorithmic insights and engineering effort. We propose to concentrate these developments around the following four research directions.

4.1 Scalable High-Level Analyses

Objective: *Develop analyses capable of propagating expressive specifications across substantial pieces of client code.*

In the Hob system we demonstrated that the use of interfaces can enable relatively simple analysis to prove important high-level application properties [69]. Our goal is to achieve similar benefits in the context of Jahob. This requires developing analyses that can extract useful information from Jahob's more expressive specification language. For this purpose, we plan to to use sound formula approximation [63, Section 4.3] and apply it to the domains of constraints supported by the analysis. We will also explore on-demand refinement of the approximation in response to detected analysis imprecision. The use of instantiable data structures in Java requires analyses that compute sufficiently precise aliasing and side-effect information. For this purpose we will explore

the use of existing analysis as well new analyses that can take advantage of any supplied invariants that specify aliasing.

We will also explore scalable solvers for new classes of constraints. As the guideline in implementing scalable constraint solvers we will explore techniques from description logic systems [106], set constraint solvers [62], and systems based on binary decision diagrams [109]. We will considered layered analysis approaches for typestate checking such as [39] as our starting point for scalable flow-sensitive analyses. We will apply our analyses to verify uses of container data structures, including iterators, to verify relationships between containers, and to detect errors such as null dereference, incorrect sequence of operations, and unreachable code.

We expect these analyses to be able to infer some of the procedure specifications. We will explore constraint simplification techniques to facilitate the communication of such inferred specifications to the developer.

Sample research subproblem: scalable solver for constraints on collections. We have established decidability and complexity results for new classes of constraints on sets and multisets with numerical constraints, which are useful for analyzing data structure clients. However, a direct implementation of these algorithms does not yield scalable solvers. Instead, we must examine constraints arising in practice and design algorithms that exploit their structure to perform well on average. This may require new algorithmic insights and implementation in the context of a theorem prover or a satisfiability modulo theory solver.

4.2 Data Structure Analysis

Objective. *Improve the performance and the scope of applicability of Jahob's data structure analysis.*

Data structure analysis is one of the current strengths of Jahob: we are aware of no other system that can verify such detailed user-supplied data structure specifications with such level of automation. Nevertheless, the precision required to verify that a real-world data structure implements the desired interface can be comparable to full program verification and still presents a challenge for the current system.

To address this challenge we will augment our current decision procedure based on monadic second-order logic with faster decision procedures, using results on less expensive logics for regular trees [45, 104] as well as generalizations of results based on small model property [7] to quantifier-free logics on trees. We will also explore more sophisticated encodings that would allow us to apply efficient decision procedures to more complex data structures. We will develop a library of transfer functions for common data structure manipulation patterns, with the goal of avoiding most of the decision procedure invocations during the analysis. With these improvements we expect to be able to verify expressive specifications of data structures present in widely used collection libraries.

Sample research sub-problem: safe data structure analysis extensions. We will explore approaches for incorporating extensions into Jahob's data structure analysis. An extension would specify a component of the analysis domain and prove the correctness of data-flow analysis transfer functions. This approach should be a generalization of Jahob's current constructs for introducing and defining specification variables. It should be possible to specify such transfer functions not only for individual statements but also for sequences of statements at once. It might be fruitful to consider the interaction of such approach with current lemma matching [63, Section 4.2.4] and semantic caching [112, Section 5] in Jahob. We will consider automation of the inference of transfer

functions using Jahob's provers as well as syntactic approaches [93, 97].

4.3 Methodology for Modular Verification

Objective. *Develop flexible and statically tractable modular verification methodology.*

To enable verification of detailed properties in large code bases, we will combine the high-level analysis with the data structure analysis. Such a combination requires a methodology for modular analysis that enforces encapsulation and is sound in the presence of mutation, aliasing, and call-backs. One design decision is the treatment of encapsulated objects, which affects the meaning of frame conditions in clients.

In the first approach, encapsulated objects are never hidden from the clients of the module, they are simply characterized in a certain way (for example, by being in a particular typestate). Although the modifies clauses do not explicitly mention changes to those objects, such changes are implicitly expanded in the client. Consequently, to prove that a method call does not modify a given object (even if the method has an empty modifies clause) it is first necessary to prove that the object cannot belong to the set of implicitly modified objects, which is a semantic check. This is the approach taken by the Spec# methodology [10].

In the second approach, the references to encapsulated objects are entirely hidden from clients. Consequently, such client can never observe the objects, the expanded modifies clauses need not mention them and the clients can often syntactically conclude that certain parts of state remain unchanged. However, this approach can be more restrictive and it is more difficult to ensure its soundness because it relies on the fact that clients can only access their reachable objects. This approach is taken by many ownership types systems [26].

We plan to develop a generic mechanism for specifying modular analysis methodologies while ensuring soundness. Using this mechanism, we will explore new points in the design space of encapsulation disciplines and compare them to recently proposed approaches in terms of programmer flexibility and the complexity of generated proof obligations. We will also explore the possibility of multiple encapsulation disciplines coexisting in the same piece of software. We anticipate that this will require module interfaces that go beyond standard preconditions and postconditions and impose requirements on all statements executing outside the module.

Sample research sub-problem: encapsulation analysis. We will develop specialized analyses that can establish proof obligations arising from frame conditions of instantiable structures, and from the encapsulation requirements for data structures. Such analysis will need to examine the invariants to determine regions of state to which they refer. This problem is in general difficult but we expect it to be solvable with precision sufficient for this purpose.

4.4 Extending the Applicability of the System

Objective. *Enable Jahob to be used by both experts and verification novices and to be applied to systems described in different languages.*

Among the most common problems in verification are errors in specifications. We will develop runtime checking techniques that allow specifications to be executed at runtime and therefore tested in a way similar to code testing. Users of the system will thus obtain concrete counterexamples for their specification errors. To enable specifications to be both tested and verified statically, we will develop formula transformations that make a large class of formulas amenable to both

approaches. In our experience with both an interpreter [116] and a runtime compiler [85] for complex specifications, pure dynamic execution yields large performance overheads. We therefore plan to explore compiler optimizations and fine-grained combinations between static and dynamic analysis to reduce this overhead. Orthogonally to these approaches, we will explore the ability to perform checks only with certain frequency and to use additional hardware resources in multi-core processors to parallelize the execution of runtime checks.

The advantage of modular verification is that each procedure can be verified in isolation. To provide the benefits of modularity to run time checking as well, we will integrate Jahob with solvers for imperative predicates [79] and declarative specifications [105]. These tools will enable generation of states that satisfy the given precondition, saving the developers the work needed to generate unit tests. Similarly to static analyses, it may be necessary to specialize these approaches to particular classes of properties to make them feasible.

In addition to the current Java front end we will explore alternative front ends for Jahob, such as annotated Java bytecodes, the Scala programming language, and the BoogiePL verification format [34]. This will generate a new set of benchmarks for Jahob and increase the number of potential Jahob users. We will also make Jahob input suitable for verifying constraints from graphical modelling notation, enabling its use not only during code development but also during software design.

Sample research sub-problem: automated and interactive proof along with runtime checking. We will extend Jahob's annotation language to optionally provide users with complete control over the proving process and thereby support verification of arbitrarily complicated properties. We will build on the current constructs such as `noteThat`. We will additionally provide support for proof rules involving quantifiers and propositional connectives, making the system complete for first-order logic. Furthermore, we will explore the interaction of such annotation language with runtime checking, with the possibility of manual proof decomposition and runtime checking simultaneously helping the verification process. We will establish connections of such approach with LCF-style theorem proving [88] and deductive runtime certification [6].

5 Time Table and Milestones

This section presents the tentative time-line for the project, the approximate assignment of objectives to doctoral students, and the results expected after each year.

We ask for the support of two doctoral students, which will work jointly with the applicant towards the goals described in Section 4. We expect the first doctoral student (Student A) to focus on the high-level analysis and modular analysis methodology. We expect the second doctoral students (Student B) to explore advances in verifying data structures, annotation-based proof system, and the integration of specification-based testing.

Year 1. The students will become familiar with the goals and the background. They will make the first steps towards the overall goals. We then expect to crystalize the modular methodology and enable modular proofs of complex properties given sufficient user annotations.

> Student A) Get acquainted with Jahob and with existing modular verification methodologies. Design and implement a generic mechanism for specifying encapsulation policies.
>
> Student B) Get acquainted with data structure verification in Jahob, with interactive program verifiers, and with interactive theorem provers. Develop an annotation-based proof system.

Year 2. We expect the main algorithmic advances in this year, which will increase the automation of data structure verification. In parallel, we will develop the enabling technology for scalable high-level analysis. At this stage, the system will be effective for verifying complex properties for programs that are properly annotated at method boundaries.

Student A) Develop core infrastructure for scalable analyses, including the front-end and the constraint solver.

Student B) Implement and evaluate the annotation-based proof system. Integrate faster automated data structure reasoning procedures into Jahob.

Year 3. We will integrate the system with specification-based testing, which will help users debug specifications and the code. We will develop and evaluate high-level analysis and apply it to a range of problems. These developments will make the system easier to use on larger code bases.

Student A) Develop high-level analysis for Jahob and apply it to detect interface usage vioations and run-time errors.

Student B) Integrate specification-based testing with static verification and interactive proof.

6 Significance and Impact of the Proposed Research

The proposed research aims to advance the principles and the practice of automated software analysis and verification.

The proposed research addresses some of the core problems of programming languages, software engineering, and automated reasoning. We will develop and implement new algorithms for deciding and solving logical constraints and synthesizing program invariants. We will establish connections between static verification, specification-based testing, and run-time checking. We will explore the impact of programming methodology on our ability to enforce that software satisfies the desired properties. Our results will therefore improve our ability to reason about software. Given that software projects are one of the largest engineered artifacts, these results will ultimately help in processing other forms of structured and semi-structured data.

We expect our results to substantially improve the effectiveness of software verification tools. All of the goals of this proposal contribute to improving the usability of Jahob: scalable analyses enable its application to larger code bases, improved data structure analysis enables verification of realistic data structures, and modular verification methodology is essential for scalability and wide applicability. Finally, techniques such as runtime checking and specification-based testing are specifically aimed at providing feedback to software developers and making it easier to use.

We will implement our contributions in the context of a publicly available distribution of the Jahob system. Jahob was already used by students in the class taught by the applicant. Once the proposed research is completed we also expect software companies and open source community to be able to use our tools and techniques to improve the reliability of produced software. Given the high cost of software errors [94], such improvements can have substantial economic benefits.

References

[1] Jean-Raymond Abrial, Matthew K. O. Lee, Dave Neilson, P. N. Scharbach, and Ib Srensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2*, pages 398–405. Springer-Verlag, 1991.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] Alexander Aiken and Manuel Fahndrich. Program analysis using mixed term and set constraints. In *Fourth International Static Analyses Symposium (SAS'97)*, 1997.

[4] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st ACM POPL*, pages 163–173, New York, NY, 1994.

[5] L. O. Andersen. *Program Analysis and Specialization of the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[6] K. Arkoudas and M. Rinard. Deductive runtime certification. In *Proceedings of the 2004 Workshop on Runtime Verification (RV 2004)*, Barcelona, Spain, April 2004.

[7] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis of single-parent heaps. In *VMCAI*, Lect. Notes in Comp. Sci. Springer, 2007.

[8] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[9] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.

[10] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[11] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[12] Mike Barnett and Wolfram Schulte. Contracts, components, and their runtime verification on the .net platform. Technical Report MSR-TR-2002-38, Microsoft Research, April 2002.

[13] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, 2004.

[14] David Basin and Stefan Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[15] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[16] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI 2003*, 2003.

[17] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In T. Ball and R.B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006, Seattle, WA, August 16-20)*, LNCS 4144, pages 532–546. Springer-Verlag, Berlin, 2006.

[18] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in a data structure verification system. In *VMCAI'07*, November 2007.

[19] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[20] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

[21] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991.

[22] Patrice Chalin, Clément Hurlin, and Joe Kiniry. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *Proceedings of Verified Software: Tools, Technologies, and Experiences (VSTTE)*, 2005.

[23] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003.

[24] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *Model Computation*, 2003.

[25] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.

[26] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proc. 13th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[27] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In *Handbook of Automated Reasoning (Volume 2)*, chapter 24. Elsevier and The MIT Press, 2001.

[28] David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting ESC/Java and JML. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[30] Bent Dandanell. Rigorous development using RAISE. In *Proceedings of the conference on Software for citical systems*, pages 29–43. ACM Press, 1991.

[31] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002.

[32] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *CADE*, 2007.

[33] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Cambridge University Press, 1998.

[34] Robert DeLine and K. Rustan M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[35] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[36] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[37] S. Feferman and R. L. Vaught. The first order properties of products of algebraic systems. *Fundamenta Mathematicae*, 47:57–103, 1959.

[38] Jean-Christophe Filliatre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[39] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. Effective typestate verification in the presence of aliasing. In *ISSTA'06*, 2006.

[40] Cormac Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.

[41] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[42] Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.

[43] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proc. 29th ACM POPL*, 2002.

[44] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. ACM PLDI*, 1991.

[45] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 342–351, New York, NY, USA, 2007. ACM Press.

[46] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

[47] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. Report on Gypsy 2.05. Technical report, University of Texas at Austin, February 1986.

[48] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.

[49] John Guttag and James Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[50] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proc. ACM PLDI*, 2001.

[51] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.

[52] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, 2002.

[53] Daniel Jackson. *Software Abstractions: Logic, Language, & Analysis*. MIT Press, 2006.

[54] Bart P. F. Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Nijmegen Institute of Computing and Information Sciences, September 2003.

[55] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Vcegar: Verilog counterexample guided abstraction refinement. In *TACAS*, 2007.

[56] He Jifeng, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In *ESOP'86*, volume 213 of *LNCS*, 1986.

[57] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall International (UK) Ltd., 1986. `http://www.vdmbook.com/jones90.pdf`.

[58] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[59] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[60] James Cornelius King. *A Program Verifier*. PhD thesis, CMU, 1970.

[61] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.

[62] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

[63] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[64] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), December 2006.

[65] Viktor Kuncak, Hai Huu Nguyen, and Martin Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006.
`http://dx.doi.org/10.1007/s10817-006-9042-1`.

[66] Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *Conference on Automateded Deduction (CADE-21)*, 2007.

[67] Patrick Lam, Viktor Kuncak, and Martin Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.

[68] Patrick Lam, Viktor Kuncak, and Martin Rinard. Cross-cutting techniques in program specification and analysis. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[69] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *6th Int. Conf. Verification, Model Checking and Abstract Interpretation*, 2005.

[70] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

[71] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proc. ACM PLDI*, 2002.

[72] Tal Lev-Ami. TVLA: A framework for Kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

[73] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. Software Testing and Analysis*, 2000.

[74] Hanbing Liu and J. Strother Moore. Java program verification via a jvm deep embedding in acl2. In *TPHOLs*, pages 184–200, 2004.

[75] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In *CAV*, 2005.

[76] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 2003.

[77] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In *IJCAR*, 2004.

[78] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2 edition, 1997.

[79] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *International Conference on Software Engineering*, 2007.

[80] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[81] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in hoare type theory. In *ESOP*, 2007.

[82] David A. Naumann and Michael Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS*, pages 313–323, 2004.

[83] Greg Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.

[84] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape, size and bag properties via separation logic. In *VMCAI*, 2007.

[85] Huu Hai Nguyen, Viktor Kuncak, and Wei Ngan Chin. Runtime checking for separation logic. Technical Report LARA-REPORT-2007-003, EPFL, 2007.

[86] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL Tutorial Draft*, March 8 2002.

[87] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[88] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. CUP, 1987.

[89] Ruzica Piskac and Viktor Kuncak. Decision procedures for multisets with cardinality constraints. Technical Report LARA-REPORT-2007-002, EPFL, 2007.

[90] Andreas Podelski and Thomas Wies. Boolean heaps. In *Proc. Int. Static Analysis Symposium*, 2005.

[91] Silvio Ranise and David Deharbe. Applying light-weight theorem proving to debugging and verifying pointer programs. In *Proc. of the 4th Workshop on First-Order Theorem Proving*, 2003.

[92] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[93] Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.

[94] RTI. The economic impacts of inadequate infrastructure for software testing. Technical Report Planing Report 02-3, National Institute of Standards and Technology, U.S. Department of Commerce, May 2002. `http://www.nist.gov/public_affairs/releases/n02-10.htm`.

[95] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.

[96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[97] Erika Rice Scherpelz, Sorin Lerner, and Craig Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *PLDI*, pages 135–145, 2007.

[98] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[99] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[100] Natarajan Shankar. Using decision procedures with a higher-order logic. In *Proc. 2001 International Conference on Theorem Proving in Higher Order Logics*, 2001.

[101] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd ACM POPL*, St. Petersburg Beach, FL, January 1996.

[102] G. Sutcliffe and C. B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[103] Mana Taghdiri. Inferring specifications to detect errors in code. In *ASE'04*, 2004.

[104] Yoshinori Tanabe, Koichi Takahashi, Mitsuharu Yamamoto, Akihiko Tozawa, and Masami Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *TABLEAUX*, pages 277–291, 2005.

[105] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

[106] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[107] Andrei Voronkov. The anatomy of Vampire (implementing bottom-up procedures with code trees). *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[108] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[109] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. ACM PLDI*, 2004.

[110] Thomas Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, September 2004.

[111] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[112] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. Verifying complex properties using symbolic shape analysis. In *Workshop on Heap Abstraction and Verification (collocated with ETAPS)*, 2007.

[113] Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[114] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004.

[115] Greta Yorsh, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 8(1), 2007.

[116] Karen Zee, Viktor Kuncak, Michael B. Taylor, and Martin Rinard. Runtime checking for program verification systems. In *Workshop on Workshop on Runtime Verification (collocated with AOSD)*, 2007.

[117] Karen Zee, Patrick Lam, Viktor Kuncak, and Martin Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.