

Specification and Verification for Future Programmers

(Details of Research Proposal)

1 Aim

The purpose of this project is to develop a methodology for the specification and verification of heap-manipulating software programs. This methodology will support an engineering approach to software development, whereby formal specifications can be used to guide and verify code implementations. The project will focus on three key aspects, namely (i) *expressivity*, (ii) *scalability* and (iii) *useability*, which are important for promoting the methodology to future programmers. Expressivity of specification will allow us to capture full functional correctness of our programs, where desired. Scalability is important to ensure that automated verification can be effectively carried out using present day's computational platforms. Useability of the technology ensures that creativity of programmers could be enhanced rather than hindered by the use of our tools. As a novel test-bed, we shall deploy the methodology and tools developed in this project for the teaching an advanced course on data structures and algorithms. This step will act as a concrete goal for our research work and will also allow the effectiveness of our methodology to be properly evaluated.

2 Significance

The project's underlying objective is to bring software verification methodology to the next level of practical use. At the present moment, most verification tools are only able to effectively handle small programs and fairly simple specifications. This status quo has to change if verification technology are to become more widely adopted. We plan to overcome three significant challenges in this endeavour.

The first challenge is to tackle the *expressivity problem* of specification. We intend to identify important abstract properties that are necessary to allow full functional correctness, where possible, to be attained for software programs. This problem is harder than it sounds, since we would have to identify (or design) a collection of provers that are up to the task in automatically verifying these abstract properties. Through expressive specifications, our methodology could now be used to guide programmers in developing programs that are guaranteed to be correct by construction. Such guarantees for software products are almost unheard of in current market practices. However, this scenario may be set for change when greater emphasis are placed in meeting the highest level of assurance (which requires verified software) as set down by the Common Criteria for Information Technology and Security Evaluation Standard (<http://csrc.nist.gov/cc/>).

The second challenge is to ensure that automated verification technology can scale to more realistic programs. We refer to this as achieving *scalable performance*. This problem is challenging since theorem proving employed by verification is itself an inherently hard computational problem. A novel mechanism we intend to design is "modular decomposition of specification" which we believe is a natural way to scale up software verification. We would also like to leverage our chances of success from the next generation of hardware technology that are based on multi-core technology. We believe that automated verification could be a serious application that benefits greatly from the current trend for parallelism through multi-core technology.

The third challenge is to ensure better *useability* of the verification technology. Here, there are two long standing issues that require attention. The first issue is better automation, particularly for finding static property of each loop, called loop invariant. There is already great progress in this area, especially for the numerical domain. Our research objective is to explore some new results for

fix-point analysis for the other more abstract domains, such as sets and maps, in conjunction with the verification framework based on separation logic. The second issue is how to better interpret a proof failure during program verification to allow program errors (if any) to be pinned down more precisely. We hope to provide a symbolic debugger for this task. As far as the PI is aware, no such symbolic debugger has ever been built to co-exist with software verification.

On a more general note, software verification has been identified as a grand challenge for computer science research [11], and the research community has committed itself to seeing “verified software” within the next two decades (vstte.ethz.ch/report.html). Our project’s objective is directly linked to this grand challenge, and we hope to make major inroads in this important direction.

3 Approach

In recent years, separation logic formalism has been successfully applied to analysing and verifying heap-manipulating programs. This formalism can support succinct description of shapely data structures, such as doubly-linked lists, circular lists, near-balanced AVL-trees [14], and can be used to analyse and verify programs that manipulate linked data structures. One important feature of separation logic is its good support for accurate references into memory states which can be captured with the help of inductive predicate (for abstraction) and separating conjunction (for disjoint memory states). For example, a list segment of length n can be specified by the following inductive predicate definition:

$$\text{lseg}(\text{root}, \text{p}, \text{n}) \equiv \text{root}=\text{p} \wedge \text{n}=0 \vee \exists \text{q} \cdot \text{root} \mapsto \text{node}(_, \text{q}) * \text{lseg}(\text{q}, \text{p}, \text{n}-1) \text{ inv } \text{n} \geq 0$$

where `node` is an object type declared as

```
data node { int val; node next }
```

In separation logic notation, $x \mapsto \text{node}(a, b)$ captures a distinct memory cell referenced from x , while separating conjunction $\Delta_1 * \Delta_2$ captures two *disjoint* heaps described by Δ_1 and Δ_2 , respectively. In contrast, $\Delta_3 \wedge \Delta_4$ captures two *overlapping* heaps, Δ_3 and Δ_4 . The formula $\text{n} \geq 0$ after the `inv` keyword captures a pure invariant for the predicate that is not dependent on the heap. This invariant is an approximation of the predicate, in the sense that whenever the predicate holds, the invariant property also holds.

Inductive predicates can also be used to describe more complex data structures with stronger properties. An example is the non-empty sorted list:

$$\begin{aligned} \text{sortl}(\text{root}, \text{n}, \text{s}, \text{b}) \equiv & \text{root} \mapsto \text{node}(\text{s}, \text{null}) \wedge \text{s} = \text{b} \wedge \text{n} = 1 \\ & \vee \exists \text{p}, \text{t} \cdot \text{root} \mapsto \text{node}(\text{s}, \text{p}) * \text{sortl}(\text{root}, \text{n}-1, \text{t}, \text{b}) \wedge \text{s} \leq \text{t} \text{ inv } \text{n} > 0 \wedge \text{s} \leq \text{b} \end{aligned}$$

Predicate `sortl`(n, s, b) ensures that all values in the list are sorted in ascending order, with s and b to capture its min and max values, respectively. The sortedness property is ensured by the presence of $s \leq t$ in the above predicate. The parameters n, s, b actually capture some derived properties of the heap data structure, as captured in the predicate definition. These parameters play a role similar to “model fields” in some specification languages, such as Spec# [12, 1] and JML [3]. However, model fields currently have to be manually instantiated by programmers, while our approach aims to automatically instantiate the predicate’s parameters by symbolic reasoning.

Though separation logic with inductive predicates can be highly expressive, current automated theorem provers hardly provide any support for this form of substructural logic. Several recent works, such as [2, 10], have attempted to address this shortcoming by building specialised solvers that work for a fixed set of predicates (e.g. the above `lseg` without the size parameter). Our recent work has lifted two crucial limitations by supporting the automated reasoning of user-defined predicates [14], and also user-defined lemmas [13]. We are planning to ride on this momentum to help push specification and verification methodology to the next level of use; towards handling realistic

programs and towards full functional correctness, where desired. We shall be focusing on three key aspects for the methodology, namely (i) expressivity (ii) useability and (iii) scalability. These three aspects are mostly orthogonal and can be pursued separately by various members of the research team. However, it is important to properly address all these three aspects together, so as to allow our methodology for specification and verification to be practically relevant. For example, if we have succeeded in developing an expressive specification that is not scalable, its usefulness may be limited to only small programs.

Our specific research plans for these improvements to the specification and verification methodology shall be organised as five work packages, as outlined below.

3.1 Work Package 1 : Expressive Specification

This work package aims to design a specification language that is *expressive* enough to capture relevant abstract properties (such as sets, bags, lists, trees and maps) for a wide range of data structures. Our ultimate goal is to support formal reasoning (with some user assistance) that can attain full functional correctness via program verification, where desired. This attempt is inspired by a recent work by Zee, Kuncak and Rinard [18], but we intend to make use of a more concise and precise specification mechanism based on separation logic instead of the traditional Hoare logic approach. We believe that our approach can result in better useability of the software verification methodology.

A key component of this work package is the customization for a range of off-the-shelf pure provers for use by our separation logic prover. Our specification is also intended to be *concise* and *extensible*. To make specification concise, we shall provide *read-only* specifications that rely on conjunctive rather than spatial views. To support *extensible* specification, we shall design a strategy that would allow multiple provers to work cooperatively. Our design allows new abstract properties and the corresponding provers to be gradually added. Together with expressivity, we expect our specification language to provide a more seamless experience to users for getting their programs correctly verified.

The key tasks of this work package are :

- Task 1.1 *Extensible Logic* : Design a logic that can easily capture abstract properties, such as lists, trees, sets, bags and maps. Provide a linkage between external syntax and the internal representation of the logic. Allow users to define a class of pure functions to support relevant abstract properties in our specification. The resulting logic shall be supported by a range of SMT solvers and specialised theorem provers.
- Task 1.2 *Read-only specifications* : The provision of read-only specification can provide a more concise and precise mechanism for immutable data structure, beyond what is possible for mutable data structures. It can be more concise since conjunctive views typically cover more scenarios. It can be more precise since cut-points (that are similar to named pointers) in heap-states are always preserved by read-only specifications but may not be preserved by mutable specifications. Our primary task here is to design a good mechanism that combines both read-only and mutable specifications together.
- Task 1.3 *Library of Specifications* : We will endeavour to provide specifications for a library of data structure algorithms with the aim of capturing their full functional correctness, where possible. We may provide multiple specifications for each data structure to capture the different scenarios that may occur. One comprehensive package that we hope to investigate is the JDSL 2.0 which is a comprehensive data structures library developed for Java. Such a study would perhaps be a first for a formally specified and verified data structures library.

3.2 Work Package 2 : Inference Mechanisms

The success of program verification technology depends to a large extent on the degree of automation that can be harnessed. Without some automation, the verification process would be unbearably painful and such an approach could only be sustained by diehard fans. As the problem of verification is in general undecidable, it is also unrealistic to demand for complete automation. We therefore advocate for a middle-of-the-road approach, whereby users are permitted to manually declare certain aspects of the software, such as : (i) predicates to describe abstract properties of data structures, (ii) pre/post conditions for each method, (iii) lemmas to explicitly capture the relationships between two or more predicates. With these hints from the users, we expect our system to autonomously verify each method for correctness. Nevertheless, there are at least two scenarios where our system can be enhanced with inference mechanisms, as highlighted next:

- Task 2.1 *Pure Invariant* : For each predicate, there exists a pure property that holds for every occurrence of the predicate. We refer to this property as the pure invariant, and shall attempt to infer the strongest possible pure invariant by fixpoint analysis. This pure property is frequently needed to support a more complete verification. Our analysis will initially assume `true` as the first approximation to the pure invariant, and repeatedly compute a stronger approximation if possible. In case `false` is derived as the pure invariant, we shall provide a warning that points to an *infeasible* predicate. Our approach need only derive the pure invariant in conjunctive form, since a more precise disjunctive invariant can always be obtained by unfolding its predicate (one or more times) prior to converting into pure form.
- Task 2.2 *Loop Invariant* : Each loop is typically considered to be a much smaller piece of code than the method body itself. There is often expectation for verification system to automatically infer the invariant of each loop, with the help of precondition given for its method. This inference process requires a good abstraction mechanism to capture the static property of the loop invariant through fix-point analysis. While the loop invariant inference mechanism is now better understood for shape [17] and numerical domains [9] separately, there is little progress on inferring the loop invariants for the other abstract properties, such as bags, maps and trees. There is also little work on inferring loop invariants that are based on a combination of abstract properties. The primary goal of this research task is to investigate how fixpoint abstraction mechanism can be generalised to cover a wider range of abstract properties. To provide flexibility to the user, we shall provide an option for the user to supply his/her own loop invariant. Naturally, each user-supplied invariant is always checked for soundness, but could be replaced by a system-generated loop invariant should the supplied loop invariant be less precise than the inferred one.

3.3 Work Package 3 : Scalable Performance

The purpose of this work package is to develop a host of techniques that would allow automated verification methodology to scale naturally to real world applications, including the classroom setting targetted by our research project. Though we do not expect very large programs to be developed in our chosen setting, we must be able to support a class of concurrent users. Given the computationally-intensive nature of automated verification, this requirement is likely to stress our best computational resources. We shall exploit three key technologies, based on *modularity*, *parallelism* and *caching*, to realise our desire in pushing verification technology out to the real world. Three main tasks of this work package are:

- Task 3.1 *Modular Scalability* : While theorem proving technology have come a long way, they are still inherently hard computational problems. For example, the state-of-the-art SAT

solvers can now handle big formulae involving ten of thousands of boolean variables, but there is no denying that the problem itself remains NP-hard. To address this challenge, we shall exploit modularity to keep our problem size manageable. Our solution is to automatically decompose each large formula, that needs to be proven, into a number of smaller formulae so that they could be more easily proven. To support this approach, we shall devise an automatic method to decompose each large specification into a set of smaller ones, and determining any dependencies between these smaller specifications. A specification X is said to be *dependent* on another specification Y, if X needs only be verified after the Y has been successfully verified. This dependency allows us to identify dead branches from the verification of Y that can be safely ignored when dealing with the verification of X. We propose to undertake an indepth study into this modular decomposition strategy, with the aim of providing a solution that is both sound and efficient.

- Task 3.2 *Grid of Provers* : Automated verification is likely to be one of the applications that can benefit greatly from low-cost parallelism promised by today’s multi-core CPUs. There are several sources of parallelism present in our modular approach to verification. For a start, we may allow each method to be separately verified in parallel. Furthermore, if multiple specifications have been provided for a method, we can verify each pair of pre/post conditions in parallel over the same method body. Within each method, it is also possible to verify each branch of a conditional in parallel, prior to joining their results at the end of the conditional. Lastly, we can make use of parallelism to harness the diversity of pure provers that are currently available through speculative processing. Different provers have different trade-offs between efficiency and coverage. By running them in parallel, we can not only choose the fastest outcome from amongst them, but may also use a collection of provers to support fault tolerance and extended coverage. We propose to built a *grid of provers* to realise this goal, we shall explore using two possible solutions. One solution is to use a low-level mechanism, like MPI where we may have more control over the concurrency. Another solution is to rely on a software package for high-throughput distributed computing, such as Condor [16], that can simultaneously manage a collection of processors for use by our pool of proving jobs.
- Task 3.3 *Proof Caching* : It is occasionally cheaper to retrieve a previous result from memory, rather than re-computing it. This is the key idea behind algorithms that are based on dynamic programming (or memo-function) that may re-use computed results that have been systematically cached in memory. Our proof caching mechanism shall exploit low-cost memory and low-cost parallelism to make this idea practical. For a start, we shall cache two main categories of proofs, namely (i) expensive proofs, and (ii) manual proofs. For proofs that are fast to compute (below a given time threshold), we shall rely on pure provers to handle them automatically, since they are cheap to re-compute. For manual proofs, it is useful to cache them, so as to extend the automated capability of our proof system should these proofs re-occur under a different context. To support fast retrieval of proofs, we shall also design good *normalization* and *indexing* mechanisms for use when caching proofs. In addition, we shall exploit inherent parallelism that is possible from the storage and retrieval of cached proofs.

3.4 Work Package 4 : Tool Development

The success of our project is dependent on the development of a good set of tools that can support automated program verification used during small-scale program development. We shall leverage our efforts on two key sub-systems that our research group have built recently, namely (i) SLEEK - a separation logic prover, and (ii) FIX - a disjunctive fixpoint calculator for numerical domain. These two sub-systems would have to be extended to incorporate better expressivity and scalability,

but our previous experience in building them have provided a major advantage to our current efforts in this project. Two key tools that we plan to construct for this work package are:

- Task 5.1 *Verifier for a Programming Language* : Our task here is to built a verifier for an imperative programming language. We will likely adopt Java as our targetted language platform, though it should also be straightforward to customize our verification technology to a different language platform. To keep our task at a manageable level, we shall employ two simplifications. Firstly, we shall focus on only static method calls and their corresponding data structures. Secondly, we shall focus on verifying only sequential programs. However, to keep our verification technology relevant to a reasonably large class of programs, we shall ensure that advanced language features, such as exceptions, genericity and arrays, are properly handled. These features require new investigations to ensure that they are properly modelled in our logic and are soundly verified.
- Task 5.2 *A Symbolic Debugger* : While the primary role of the program verifier is to certify that user programs are safe in meeting the obligations of their stated specifications, there is also a need to pin-point the source of a potential programming error should a given proof obligation fails. We propose a symbolic debugger to help in this process. Unlike runtime debuggers which execute on concrete input values, our debugger shall operate over symbolic values with each program state captured by a separation logic formula. We can identify a set of program paths that directly contribute towards each failed proof by precisely associating each program path with its respective heap state. Program paths that do not contribute to the failed proof can be omitted from consideration. Our debugger may then display a sequence of heap states for each program point of the chosen program path leading to the failed proof. Suitable animated display can help us more easily identify statements that are likely causes of the failed proof.

3.5 Work Package 5 : Classroom Experience

A tangible goal we have set for our project is the use of our research tool to aid in the teaching of a data structures and algorithms course to a special class of computer science students. This special class is targetted at bright students with good academic credentials. We believe that this novel attempt can be both rigorous and stimulating to these computer science students, allowing them to appreciate the importance of specification and design early in their programming career. Though this teaching is not strictly part of the research per se, the early user experience from deploying our tools could be immensely important towards a review on the next generation of toolkits needed for the software verification methodology. Two key tasks of this work package are:

- Task 6.1 *Course Teaching* : The specification and verification methodology shall be used to augment the teaching for a special class on data structures and algorithm. The students will be taught how to formally specify data structures, write pre/post annotations for each method in addition to the design of data structures and their corresponding algorithms. They will also be taught how to trace through their program's execution with the help of our symbolic debugger. Relevant experiments and assessment materials will be prepared for the course. We shall also be developing the course material and instruction package to support the teaching of data structures and algorithms with the help of specification and verification. A set of course notes will also be written.
- Task 6.2 *Evaluation* : We shall carry out regular evaluation (that includes assessments and surveys) to determine the benefits of using both specification and verification in teaching an advanced programming course. The main purpose of this evaluation is to collect feedbacks

from users on our developed tools. This is expected to allow us to make both incremental and radical improvements to our tool set. The feedback loop is expected to make our tool more precise and practically relevant.

4 Investigators

The PI A/P Chin Wei Ngan will be involved in all phases of this project and will spend 60% of his research time on this project. Two collaborators A/P Khoo Siau Cheng and A/P Dong Jinsong will contribute about 10% of their research time each in this project.

A/P Khoo has worked closely with the PI on several past projects, including a project on calculating sized type. This past expertise will be very useful for formalising the inference work package. His more recent research interests have been on dynamic program analyses techniques, and we hope to tap this expertise to design a new symbolic debugger based on separation logic.

A/P Dong has previously worked on extending the Z specification language, and is recently involved in projects on model-checking techniques. He is also teaching a graduate course on specification mechanisms. We are hoping to tap his expertise to help with the work packages on expressive specification and classroom experience.

The project has provisions to invite two overseas collaborators as Visiting Professor/Fellow.

Prof Martin Rinard from MIT has previously collaborated with our group under the Singapore-MIT Alliance program. He has made regular visits to Singapore in the past and we have always appreciated his invaluable advices on various aspects of our research work. His team's recent work on JaHob system (reported at PLDI 2008) uses Hoare-style verification to explore full functional correctness for linked data structures. Prof Rinard will act as a consultant for our project. During each of his annual visit, we will organise an informal workshop to present the progress of our research, and will solicit valuable advices/feedbacks/exchanges during these workshop sessions.

Dr Shengchao Qin from University of Durham has been working with the PI in an earlier project on automated verification via separation logic. For his next major project (which was recently approved by EPSRC,UK to start in Sep 2009), Dr Qin is planning to investigate inference mechanisms for separation logic in collaboration with our group. His visit here will help us provide a focus on the work packages dealing with inference mechanism. We will jointly explore new inference mechanisms suitable for the toolset developed by our group in NUS. During Dr Qin's visit, we plan to engage him as a Senior Research Fellow (for short periods) to allow him to formally collaborate with us on this project.

5 Environment

The project will be carried out by a team led by the A/P Chin and A/P Khoo from the Programming Languages and Systems Research Laboratory in Dept of Computer Science, National University of Singapore, in collaboration with the group led by A/P Dong from the Software Engineering Research Laboratory. We expect synergistic result from this collaboration whereby practical approach from software engineering methodology could be fused with foundational approach to verification from the programming language perspective.

The PI currently has two very good PhD students working on related research topics. One of them is in the first year, while the other is in the third year of PhD. The two students will provide the initial impetus and efforts to kick off the research project, and we expect to recruit new PhD students and research assistants to ramp up the project from there.

6 Preliminary Studies / Progress Reports

Our proposal for an expressive specification and verification methodology will exploit key technologies that we have developed in recent years. For the past five years, we have accumulated significant amount of experience in both the design and the implementation of a variety of static program analyses systems based on advanced types [4, 6, 8], generic types for OO language [5], disjunctive fix-point analysis [15] and have started preliminary works on automated verification that are based on separation logic with support for user-defined lemmas and predicates [14, 7, 13]. Our work have been funded by two main sources (i) a S\$438K grant A*STAR on “A Constructive Approach to Dependable Software”, (ii) three small grants from Microsoft Research totaling S\$125K on the themes of trustworthy computing and type-based analyses and a small IBM Innovation award grant of about S\$15K. These prior projects have enabled us to built several research prototypes to demonstrate our tools’ capability at handling a variety of program analysis, optimisation and type-checking problems. Three of our recently developed research prototypes are considered most relevant to our new research project proposal, as described below:

1. HIPO : This is one of the first working verification tool for the object-oriented language paradigm that is based on separation logic. To strike a balance between precision and reuse, our verifier supports both a static and a dynamic specification for each method. The former is for statically-dispatched calls, while the latter is for dynamically-dispatched calls. Our prototype verifier currently handles only small OO programs. This result was recently presented at the ACM POPL 2008 conference The tool is accessible from :
<http://loris-7.ddns.comp.nus.edu.sg/~project/hip/index.html>
2. SLEEK : This is our entailment prover for separation logic with support for linear arithmetic, set and bag properties. Our prover pioneered several new features, including automatic frame-infering capability, support for user-defined predicates/lemmas, and also integration with off-the-shelf pure provers. Our results have been presented at both VMCAI 2007 and CAV 2008 conferences. The tool is accessible from :
<http://loris-7.ddns.comp.nus.edu.sg/~project/sleek/index.html>
3. FIX : This is a fixpoint analysis tool for disjunctive numerical domains. It uses an affinity heuristic that can give a good trade-off between precision and scalability of inference mechanism. The work has been reported in ASIAN 2006 conference, and can give very accurate program analysis results though it is presently limited to linear arithmetic domain. We intend to extend the fix-point analysis tool to other abstract domains in the new project. The tool is accessible from <http://loris-7.ddns.comp.nus.edu.sg/~project/fixpoint/index.html>

These systems are research prototypes that would allow us to built the next generation of tools needed for the methodology of specification and automated verification. While the existing prototypes can be used to *demonstrate* the feasibility of automated verification for small examples, they cannot yet handle realistic programs as they have not considered key challenges that have been identified in this proposal, namely (i) expressive specification, (ii) scalable performance, and (iii) better useability. Our new research project will propose significant extensions to the specification language which will necessitate the adoption of new pure provers and the design of new techniques for handling the more expressive logic. Our project will also provide new mechanisms to ensure that automated verification can be scaled up to handle realistic programs by exploring the use of modular decomposition, grid of provers, and proof caching. With our past research track records, we believe that we are well positioned to take automated software verification methodology to the next level of use by future programmers.

References

- [1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362, pages 49–69. Springer-Verlag, LNCS, 2004.
- [2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780, pages 52–68. Springer-Verlag, November 2005.
- [3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.
- [4] W. N. Chin, S. C. Khoo, and D. N. Xu. Extending sized type with collection analysis. In *ACM SIGPLAN PEPM*, pages 75–84. ACM Press, 2003.
- [5] Wei-Ngan Chin, Florin Craciun, Siau-Cheng Khoo, and Corneliu Popeea. A flow-based approach for variant parametric types. In *OOPSLA*, pages 273–290, 2006.
- [6] W.N. Chin, F. Craciun, S.C. Qin, and M. Rinard. Region Inference for an Object-Oriented Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Washington, D.C., June 2004.
- [7] W.N. Chin, C David, H.H Nguyen, and S.C. Qin. Enhancing modular OO verification with separation logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 87–99, 2008.
- [8] W.N. Chin, S.C. Khoo, S.C. Qin, C. Popeea, and H.H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. In *ACM SIGSOFT ICSE*, pages 186–195, St. Louis, Missouri, May 2005.
- [9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1977.
- [10] A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, Springer LNCS, pages 240–260, Seoul, Korea, August 2006.
- [11] Cliff B. Jones, Peter W. O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.
- [12] K.R.M. Leino and P. Muller. A verification methodology for model fields. In *15th ESOP*, March 2006.
- [13] H.H. Nguyen and W.N. Chin. Enhancing program verification with lemmas. In *Intl Conference on Computer Aided Verification (CAV)*, 2008.
- [14] H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, Nice, France, January 2007.
- [15] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2006.
- [16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.
- [17] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.
- [18] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.