Abstraction-carrying Code: a New Method to Certify Temporal Properties

Songtao Xia and James Hook

OGI School of Science & Engineering
Oregon Health & Science University
Beaverton OR 97006, USA

Abstract. Temporal properties represent an important class of properties that, when verified, help to establish high confidence of a program. Certification of temporal properties provides such confidence to a client of a mobile program. Current techniques adopt a proof of a temporal property as a certificate; they therefore suffer from the state-explosion problem. Using these techniques, we may not afford the certification of multiple properties.

We propose abstraction-carrying code (ACC). This approach adopts an abstract interpretation of a program as a certificate. A client will validate and model check the abstraction to verify the temporal properties.

This paper provides an overview of ACC. We summarize our early results, which include 1) how we generate an abstraction for an intermediate program, 2) how we represent an abstraction in an easily verifiable way, and 3) experience of application of ACC to a few industrial strength problems. The paper also discusses new perspectives made possible by the ACC framework.

1 Introduction

In the motivating scenario the developer of a mobile application develops software anticipating remote verification. The software is packaged with checkable annotations. These annotations are communicated with the software to the client, who uses the annotations to validate required temporal properties. These properties range from data-flow safety properties, such as the correct use of APIs, to liveness properties.

Research on proof-carrying code, initiated by Necula and Lee, addresses the problem of how a client trusts an untrusted program by verifying a mathematical proof for certain safety properties [17]. Early variants of PCC, including TAL [16, 15], DTAL [21], ECC [13], Certified Binary [19], FPCC [1], TouchStone and others, focus primarily on type safety and memory safety.

Researchers have begun to explore extending PCC to certify temporal properties. For example, Temporal PCC (TPCC) is able to certify and verify temporal properties [3]. It represent the traditional line of thinking inherited from PCC, which is catch-phrased by "proof as certificate". The success of PCC may have allured people to work on proofs when certificates are needed. For example, systems are proposed, and in some case implemented, to construct a proof from the result of a model checker [11, 12, 2, 20].

These proof-based certification methods all suffer from the state explosion problem. The size of a proof tends to expand drastically as the problem size grows. Because of this problem, we cannot cash in a lot of applications made possible by the expressive power of temporal properties. Examples include fully automated certification of general temporal properties, including liveness properties, of concurrent programs. Because of the same problem, with limited network bandwith, a server may not be able to certify multiple properties at one time, which is one of most demanding requirement in a realistic system. Furthermore, in a proof based certification system, the certificate is tightly coupled with the property of concern: a proof for one property proves that property only. This fact makes it impossible to keep a server from knowing the exact security policy running by a client.

The approach presented here, partially inspired by SLAM, Bandera and other related projects, adopts predicate abstraction as the certification method. More specifically, on the server side, we adopt Boolean abstraction [8, 5, 2] to compute an abstract model of a source program. Then we compile the source program, in an abstraction-preserving fashion, to intermediate program. The result of the compilation is an intermediate program that has the same Boolean abstraction as the source program. The server will send the boolean programs, encoded as index type annotations, as the certificate to the client. The client will use an index type checker to validate the abstraction before it model checks the temporal properties on the abstract model. Index types are a version of refinement types that augment conventional type checking with a constraint satisfaction mechanism providing some of the expressive power of dependent types. In one index type system, not only can we verify the abstract model but we also verify system invariants. We have implemented a prototype infrastructure of ACC in a toolkit called ACCEPT (ACC Evaluation Prototype Toolkit). An illustration of the infrastructure is in Fig.1.

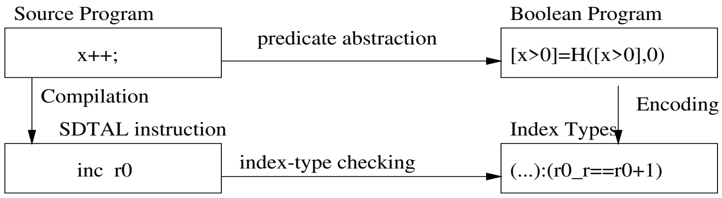

Fig. 1. Relation between source program, assembly program, boolean program and index type

This paper provides an overview of ACC. We summarize our early results, which include 1) how we generate an abstraction for an intermediate program,

2) how we represent an abstraction in an easily verifiable way, and 3) experience of application of ACC to a few industrial strength problems. The paper also discusses new perspectives made possible by the ACC framework.

The paper is structured as follows. In Section 2, we introduce a few examples that motivate our investigation. Next we present the framework of ACC in terms of control flow automata (CFA). In Section 4 we summarize our experience with applying two versions of ACC implementation to two categories of programs, including the certification of industrial strength programs. Certification of multiple properties is not yet implemented; we discuss related technical issues in Section 5. Finally we conclude.

2 Motivation Examples

2.1 Example 1: Concurrency

On the left side of Fig. 2 we list the code fragment of a bounded buffer program written in a Java-like language. There are two symmetric threads, the figure showing one of them. Global variables $\texttt{buf}_1$ and $\texttt{buf}_2$ count the elements in the buffers. The capacity of both buffers is a constant `bound`. The thread shown constantly moves elements from one buffer to the other. The other thread does the opposite. One of the properties to expect is that whenever a buffer is full, then sometime in the future one of the elements must be removed. This property, which we name as FullToNonFull, can be expressed in a Linear Temporal Logic (LTL) term as

$$\Box(\text{Full} \rightarrow \Diamond\text{NonFull}) \qquad \text{(FullToNonFull)}.$$

Model checking this property on the source program directly is relatively expensive due to the state explosion. A popular way to attack this problem is predicate abstraction. This abstraction method observes a program's behavior on a selected set of predicates and translates the statements of the program into statements that operates exclusively on the boolean variables representing the predicates. Bandera [4], for example, can be applied to verify this property. The problem is, can we reuse the verification work done by a tool like Bandera such that when a client download this piece of code, it does not have to run Bandera again?

The right side of Fig. 2 listed the Boolean program of the first loop body, based on the following predicates.

$$\begin{aligned} \{\text{Empty}_i &: \texttt{buf}_i = 0, \qquad \text{where } i = 0, 1. \\ \text{Full}_i &: \texttt{buf}_i = \texttt{bound}, \\ \text{NonFull}_i &: \texttt{buf}_i < \texttt{bound}\} \end{aligned}$$

.

2.2 Example 2: Multiple Properties

In reality, many things can go wrong with a program. For example, a mobile program may use a lock; it may read from and write to a file; and it may

```
// int buf1=bound, buf2=0;
// int bound= 50;

thread1 {
  int buf1 = bound;
  run() {
  while (true){
    synchronized {
      while (buf1 <= 0)
       wait();
      buf1--;};

    synchronized {
      while (buf2 >= bound)
       wait();
     buf2++;}
 }
 }
}
// thread2 symmetric to thread1
```

```
while 1
 synchronized{
    while (*) {assume(empty1);wait};
     assume (! empty1);
     [
     full1 := H(0,full1 || nonfull1),
     nonfull1 := H(full1 || nonfull1, 0),
     empty1 := *;
     ]
 }
while 1
 synchronized{
   while (*) { assume (full2); wait};
     assume (! full2);
     [
     empty2 := H(0,empty2 || nonfull2),
     nonfull2 := H(0, full2 ),
     full2 := *;
     ]
 }
```

Fig. 2. Bounded Buffer Example and Its Predicate Abstraction (Some omissions are made)

contain a few threads, as the previous example does. For a client to have higher confidence of this program, a server may certify all these properties.

Another scenario: different client may have different needs; they may enforce different security policies. A simple example is that a client may not allow network connection after the mobile program has read from the local disk while this operation is totally legitimate and necessary for another client. Typically, these operations are granted at run time, which certainly will incur some costs. Certification of relevant temporal properties will allow a client to grant or refuse certain operations at link time and save the client from these costs. The problem is: a server may not be able to tell beforehand which client wants which property.

When multiple properties are to be certified, the size of a certificate may become even more significant an issue. Using current techniques, the total size of the certificate will be too large to be transmited over the internet. In fact, this kind of problem is common for the proof-based certification techniques. Because a proof for a temporal property must deal with every possible path a system may go through, and for software, the number of such paths can be large, the size of the proof is almost inevitably big, even when the property is tested on an abstraction of the program.

For example, BLAST generates small size certificates for reachability programs. But for a single property specified on a windows driver, the size of the program can be as large as more than 100 Mega bytes. BLAST is representive in this aspect of other proof-based approaches.

3 Abstraction as Certificate

Based on the discussion above, we seek other methods to certify a temporal property. One of the obvious choices is to use an abstract interpretation of the mobile program as certificate. In theory, any kind of abstraction that enables successful model checking can serve as the certification method. However, we are going to choose Boolean abstraction. This choice has the following direct benefits.

- A large set of temporal properties can be certified.
- The size of a certificate is controled. The size of a Boolean abstraction is typically small, linear in the size of the program.
- It is feasible to implement such a system. Abstraction is computed as part of the verification result. Boolean abstraction have been successful in verifying temporal properties on industrial strength programs.
- Verification time can be reasonable. Model checking an abstraction is typically much faster than model checking the source program, if the latter is ever possible. Thus a client may not need very long time verifying the temporal property.

More interestingly, when two sets of predicates are generated for two properties, we can generate a Boolean program using the union of these two predicate sets. A client, however, can be informed of which predicates are for which property. The client can extract a Boolean program from the sum Boolean program and model check the property on it.

ACC represents a local maximal point in the design space. Comparing to proof-based approach, which generates relative larger certificates that are quicker to verify, ACC generates smaller certificates that take longer to verify. More specifically, model checking is needed by ACC to verify the temporal property, while the proof based approach simply verifies the attached proof. An important aspect in evaluating the practicality of ACC is to demonstrate that verification time is acceptable, at least for certain applications.

Another dimension of comparison is the flexibility. With proof-based approach, one proof is generated for one property; the server must know the exact property a client expects to provide a proof. To provide a choice for the client between properties ϕ_1 and ϕ_2, a server has to provide a proof of $\phi_1 \wedge \phi_2$. With ACC it is possible to provide an abstraction that may certify multiple number of properties, including ones that are not known to the server at all. Detailed discussion on this issue is in Sec 5.

Next we present the ACC framework, with brief introduction to how we have implemented this system.

3.1 Control Flow Automata

We use a control flow automaton (CFA) to model a program's control flow. Intuitively, a CFA comprises: (1) variables, such as local and global variables in a source program, or memory locations, including registers, in an intermediate program, (2) vertices: control locations of a program, and (3) labeled directed edges that connect vertices. An edge is labeled either with a sequence of instructions that are executed to move from the source to destination locations of the edge, or with a predicate corresponding to the branch condition that must be true for that edge to be taken, or with a symbol for surrendering the CPU in a concurrent setting.

Formally, a CFA is a tuple $(Q, Q_0, X, \mathtt{Op}, \rightarrow)$, where Q is a finite set of control locations, Q_0 is the set of initial control locations, X is a finite set of variables, $\mathtt{Op}$ is a set of operations, and $\rightarrow \subseteq (Q \times \mathtt{Op} \times Q)$ is a finite set of edges labeled with operations. The operation allowed is one of the following: (1) a sequence that consists of one or more instructions or statements, (2) a predicate that enables the transition, or (3) a symbol $\mathtt{i}$ that indicates the CPU may be surrendered to another thread (this case is only useful in languages with a notion of interleaved concurrency).

A program written in C can be translated into a CFA [14]. Because a Boolean program shares the same CFA as the original program, if we want to associate the intermediate program with the same Boolean program, it is essential to preserve the CFA structure when compiling the source program into an intermediate program.

3.2 From CFA to Transition System

Let $\mathfrak{v}_X$ be valuations over X that contain type-preserving functions from X to values. A state is a pair in $Q \times \mathfrak{v}_X$. A region is a set of states. Let R range over regions. The location component q in a state (q, V_X) can be viewed as part of X (technically X may need to be expanded). Thus a state is an element in $\mathfrak{v}_X$. A transition system induced by a CFA $(Q, Q_0, X, \mathtt{Op}, \rightarrow)$ will be a tuple $(\mathfrak{R}, R_0, \rightarrow)$, where $\mathfrak{R}$ here represents the set of all possible states, R_0 is the set of the initial states, and $\rightarrow$ is reused to represent the transition relation defined by edges $\rightarrow$. Note that we may refer to this transition relation as the post operator. Accepting states are not defined in this system.

Standard techniques allow us to determine if this transition system satisfies an LTL formula ϕ, provided that we can evaluate the atomic propositions of ϕ at a given state. We first construct a Büchi automaton that accepts the negation of ϕ. Then we construct a product automaton of the Büchi automaton and the transition system. Model checking this automaton will evaluate ϕ.

So far we have not distinguished a concrete program and an abstract program. An abstract program is induced by an abstraction of the state space X. The abstraction, as usual, is defined by a pair (α, γ) that forms a Galois connection between all concrete states and all abstract states. The abstraction function α will map a concrete region to an abstract one. The pair induces an abstract transition relation $\rightarrow_a$ between abstract regions. We thus can compute an abstract transition system defined by an abstract state space and $\rightarrow_a$. We can model check the abstract transition system to verify the property on the concrete system. Because of the Galois connection requirement, the relation $\rightarrow_a$ is an over approximation of $\rightarrow$. As a result, if we model check the abstract transition system, only global properties, such as LTL formulas, can be verified on the original system.

ACCEPT adopts Boolean abstraction, where the abstract transition $\rightarrow_a$ is defined by a Boolean program[2]. Usually, for a software system, most abstractions will verify a next operator-free LTL formula.

3.3 Concurrency

Assume a source statement is atomic, that is, modifying at most one shared variable[1]. We can apply ACC to concurrent programs. The Boolean program translation of a statement will correspond to an atomic block in a language such as Promela.

For every statement in a parallel thread, we generate an $\mathfrak{i}$ transition to any other statement in other thread. Such a transition may or may not be enabled during model checking. For example, the model checker may only choose the next statement of a pending thread. Even so, the introduction of this $\mathfrak{i}$ causes the model checker to consider too many choices and slows down the model checking significantly.

One way to attack this problem is through thread-modular verification, based on assume-guarantee principle [18], as implemented by a few state-of-art systems [9, 6, 7]. Although these systems have been successful, not all temporal properties can be automatically checked this way. Our approach is to analyze the concurrent primitives to reduce the number of $\mathfrak{i}$ transitions. For example, when two critical sections are protected by the same lock, and we know mutual execlusion holds, we can delete the $\mathfrak{i}$ transitions from a statement in the critical sections to statements in the other critical section. This idea is partially tested in ACCEPT by certifying liveness properties on some toy concurrent programs..

Another issue is that model checking a concurrent Boolean program (with function call) may not terminate, which, however, is not within the scope of this paper.

3.4 On Implementation of ACC

We have answered a few questions necessary to implement an ACC system. First, a mobile program is often compiled. Directly computing an abstraction of an intermediate program suffers from the fact too many predicates are needed. How do we compute a useful abstraction for an assembly or an intermediate program? Second, Is there a way to represent the certificate, a Boolean program, in an efficiently verifiable way? We will devise an abstraction-preserving compilation scheme with which we can reuse a Boolean program for the source program as a valid abstraction of an assembly program. We will use index types in an index type assembly language to encode the Boolean program. An index type type

[1] A similar assumption is anyway needed for the soundness of abstraction based model checking.

checker is invoked to validate the encoded Boolean program. These have been covered by our sister papers [23, 22].

We implemented ACC system in ACCEPT. We present our experience below.

4 Experience

We have constructed two instances of the ACCEPT framework. The first version, ACCEPT/While, implements a simple compiler for a while language with concurrency primitives. It generates programs for an index-typed JVM dialect that we developed earlier. The second version, ACCEPT/C, leverages the BLAST infrastructure, including CIL. It operates on C language programs. It also produces index-typed JVM code.

The basic components of ACCEPT include an encoder that encodes Boolean programs into type annotations is implemented together with the type checker of SDTAL.

Initial results with ACCEPT/While are reported in Table 3, which lists the size of the certificate generated for several well studied concurrent programs and some liveness properties. The percentage indicates that, even when the program does not contain much irrelevant code (which a realistic program often does), the size of the certificate is small. Also, we listed the verification time spent on model checking Promela programs translated from the abstracted programs. The measurement is done on a Pentium III 700 Hz laptop with 256MB memory running Linux.

	Program Size	Certificate Size	Percentage	Verification Time
Bounded Buffer	702	232	33%	0.42
Vector Addition	808	308	38%	0.30
Reader/Writer	1570	282	18%	0.43
Sleeping Barber	809	200	25%	0.74

Fig. 3. Size of Certificate in Bytes

Recently, we migrated BLAST [10] on top of ACCEPT infrastructure, resulting in ACCEPT/C. We are able to reuse the basic components of ACCEPT. BLAST handles the full syntax of C programs through the use of CIL [14]; BLAST adopts lazy abstraction, a variation of predicate abstraction, to verify reachability properties of industrial strength programs, such as device drivers. BLAST represents C programs as labeled transition systems built from CFAs during model checking. We compute a Boolean program from BLAST's intermediate result, after BLAST has gone through the refinement based predicate discovery process to settle down with a set of predicates with which a useful abstraction is made. Then we compile the CFA representation to JVM code and encode the Boolean program as index type annotations. On the client side, ACCEPT's SDTAL type checker will validate the Boolean program.

The lazy abstraction is a dynamically (in the process of model checking) scoped predicate abstraction. When we generate Boolean programs, we take the sum of those predicate sets that are used to abstract the same operation; the results are statically scoped predicate sets. We use these predicate sets to abstract the C program.

We are able to generate relatively small certificates. For example, for the correct use of locks, the certificates generated for most Linux drivers, whose sizes range from a few hundreds lines to hundreds of thousands lines, range from hundreds of bytes to a few thousands bytes. The figure is smaller than the size of certificates generated by BLAST. Note that BLAST only certifies source programs. We compare our result with BLAST here assuming that BLAST has adopted our compilation scheme and that the proof it generates is valid for the intermediate program.

Another way to compare with BLAST will count the number of verification conditions that a certificate induces. With BLAST, the number is the size of the search tree because a location invariant in BLAST needs regions from every node in the search tree corresponding to the location. Our certificate is linear in the size of the relevant part of the program. For correct use of locks, a search tree often collapses into a chain because of the typical way a lock is used. For such applications, BLAST is able to generate very small certificates. For more complicated properties, such as the IRP specification of Windows NT drivers, a typical proof generated by SLAM can be about 100 Mega bytes, which can be compressed to about a few Mega bytes. A typical ACCEPT/C certificate is about 100 Kilo bytes before compression.

The price we pay is the time spent on model checking. While BLAST verifies a proof of the size of the search tree, we need to generate the search tree. This means calling a decision procedure as many times as the size of the search tree in the worst case. In practice, such search doesn't take long[10], ranging from less than a second to a few seconds for checking the correct use of locks for Linux device drivers. In Table 4, we list the results of certifying a few Linux device drivers using our system. The verification time is the model checking time reported by BLAST. The numbers can be improved if a quicker model checker is adopted. The program size is the number of lines of code after we pre-process the source program. Certificate size is measured in bytes. Our experience confirmed that the verification time is acceptable for some applications.

Negative results do appear in our experience. For example, the Windows device driver examples often take more than a few hundreds of seconds to model check. When that happens, the choice between ACCEPT/C and BLAST depends on how long it takes to download the Mega-byte proof generated by BLAST. Still, the solution based on ACC may be viable in certain circumstances. For example, the client might tolerate hundred-second model checking time and the network connection is rather slow.

To summarize, using ACCEPT/C, we obtain a practical solution to the certification of reachability properties on realistic C programs. This solution trades model checking time for certificate size.

	Program Size	Our Certificate Size	BLAST Certificate Size	Verification Time
atp.c	2482	1212	8737	$1.31sec$
ide.c	$48k$	876	12452	$0.37sec$
audio.c	$175k$	15410	502575	$5.60sec$

Fig. 4. Comparison between ACCEPT/C and BLAST

5 Discussion

Adopting Boolean abstraction as the certification method, we can certify multiple properties by constructing a sum abstraction. A sum abstraction of two Boolean abstraction based on predicate set P_1 and P_2 is a predicate abstraction based on $P_1 \cup P_2$. A Boolean program based on predicate set P can be projected to a Boolean program based on Q, where $Q \subseteq P$. The projection can be recursively defined over expressions as following.

- A boolean variable b is projected to b if $b \in Q$, or * if not.
- Negation, disjunction and conjunction is recursively defined over its operands. When * is involved, $\neg * = *$; $a \wedge * = *$ and $a \vee * = a$.

It is relatively easy to extend this definition over `assume` and assignment statements in a Boolean program. Such a projection operation is expected to extract a Boolean program based on Q from the certificate. This way, we have a method to certify and selectively verify multiple properties. Note that this is however, not unique to ACC. With proof based approach, we can also extract one proof of a property from a proof for conjunctions of multiple properties. The difference is still about the size of the certificate.

Strictly speaking, just as a proof is directly linked to a property, in a counterexample guided refinement predicate discovery system, a resultant Boolean program is directly linked to a temporal property. The concept of ACC, in general, allows us to separate a server from knowing the exact policy of a client because multiple properties can be verified on one abstraction. To fully realize this potential, we need research on feed-back free predicate discovery, which we conjecture might be possible for some properties.

Another possible future work is to reduce the size of a Boolean program. In a system like SLAM or BLAST, refinement-based predicate discovery is performed to find a right set of predicates. A new predicate is introduced by applying a so-called focus operator to predicates. When this process is performed automatically, many predicates are in fact related. Binate covering is a technique commonly used in digital circuit optimization. Given a boolean function, binate covering finds a minimum representation of this function, subject to a set of constraints. In our setting, the constraints are the relation among predicates. For example, Full and NonFull in our early example are related, Full implies the opposite of NonFull. Finding a minimum representation of the post operator minimizes the size of the certificate, which in turn reduces the cost of model checking.

6 Conclusion

In this paper, we describe abstraction-carrying code, which offers a method that may generate small size of certificates for temporal properties of sequential and concurrent programs. We summarize some technical development involved in designing and implementing an ACC system. Some experiences of application of ACC are also reported. We suggest the application of ACC to the certification of multiple temporal properties, which can lead to an separation of knowledge between a client and a server. We discuss the technical issues related to this new possibility, as well as other techniques that may improve the performance of an ACC system.

7 Acknowledgement

A long list of people offered help to our research. The idea originated in a class taught by Mike Gordon two years ago and was later sharpened by discussion with Penix, Wesser, Ball, Amy Felty, Rajamani, Andrew Bernard and Hatcliff. Hatcliff also sent us a simple implementation of predicate abstraction to help us understand the technique. Special thanks to Rupak Majumdar, who provided newest version of BLAST.

Discussions with our collegues at Pacsoft, especially with John Matthews and Mark Jones, are both informative and encouraging. Thank Tim Sheard for his tutoring the first author on English writing.

References

1. A. Appel. Fundational Proof-carrying Code. In *Proceeding of 16th IEEE Symposium on Logics in Computer Science*, June 2001.
2. T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science,LNCS2057*, pages 103–122. Springer-Verlag, May 2001.
3. A. Bernard and P. Lee. Temporal Logic for Proof-carrying Code. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Danmark, July 2002.

4. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Extract Finite-state Models from Java Source Code. In *Proceedings of ICSE 2000*, 2000.
5. S. Das, D. Dill, and S. J. Park. Experience with Predicate Abstraction. In *Proceedings of CAV'99, LNCS 1633*, pages 160–171, Trento, Italy, July 1999.
6. C. Flanagan, S. Freund, and S. Qadeer. Thread modular verification for shared memory programs. In *Proceedings of ESOP 02*, 2002.
7. C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of SPIN 2003*, 2003.
8. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97, Lecture Notes in Computer Science, LNCS 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
9. T. Henzinger, R. Jhala, R. Majumdar, , and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of CAV 03*, 2003.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70, 2002.
11. Kedar S. Namjoshi. Certifying Model Checkers. In *13th Conference on Computer Aided Verification (CAV)*, 2001.
12. Kedar S. Namjoshi. Lifting Temporal Proofs through Abstractions. In *VMCAI*, 2003.
13. D. Kozen. Efficient Code Certification, 1998. Technical Report, Computer Science Department, Cornell University.
14. S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. Cil: Intermediate languages and tools for c program analysis and transformation. In *Proceedings of Conference on Compiler Construction (CC'02)*, March 2002.
15. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stacked-based Typed Assembly Language. In *Proceedings of workshop on Types in Compilation, also in Lecture Notes in Computer Science, LNCS 1473*, pages 28–52. Springer Verlag, 1998.
16. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
17. G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
18. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, pages 319–340, 1976.
19. Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. Type System for Certified Binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 217–232, January 2002.
20. L. Tan and R. Cleaveland. Evidence-based model checking. In *13th Conference on Computer Aided Verfication (CAV)*, pages 455–470, 2001.
21. H. Xi and R. Harper. Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Florence, September 2001.
22. S. Xia and J. Hook. Experience with abstraction-carrying code. In *Proceedings of Software Model Checking Workshop*, 2003.
23. S. Xia and J. Hook. An implementation of abstraction-carrying code. In *Proceedings of Foundations of Computer Security Workshop*, 2003.