C Project Description

C.1 Introduction

Our goal is to invent a class of new techniques and tools for finding and removing defects in software systems, and to improve current methodology for designing high-quality systems at the outset. Our proposal brings together expertise in formal verification and theorem proving, model checking, and program analysis. Each of these three areas has developed rapidly in recent years, seeing both significant theoretical and practical advances. It is our thesis that further advances are possible in a project that combines these areas to work on a common set of problems.

We will structure the project to simultaneously pursue experimental and theoretical avenues of research. History and our own previous experience strongly suggest that both are necessary for productive research in the software area.

C.1.1 Experimental Driver: Open Source Software

Our experimental and system-engineering efforts will be focused on designing and building tools to improve the quality of Open Source software. Open Source is extremely attractive as a research vehicle in software quality for several reasons:

- Open Source is critical infrastructure for the new economy. Most visibly, millions of web servers world-wide run on Open Source software, but additional uncounted numbers of desktop machines, mail servers, and databases run Open Source software on private networks.

- Open Source software has quality problems. While these problems are likely no worse than those for any other kind of software, reading any of the Open Source newsgroups or development mailing lists is to read a never-ending discussion of bugs.

- Open Source is available—all of it. Open Source is unique in that a complete, real-world system is available for study, including the device drivers, the operating system, the compilers, and the applications.

- Open Source is a living, changing world. A difficulty in software research is that one is often forced to work with mature, largely unchanging software, which is far from the normal state for important software systems.

- By working with Open Source, we will be able to evaluate the benefit of the tools we build. Because of the Open Source tradition of incorporating useful new techniques and tools into the Open Source environment, we also have the opportunity for direct impact.

A recent example of work involving members of the project illustrates the style of research, type of results, and potential impact of working with Open Source. *Buffer overruns* are a common error in C programs. Because C lacks array bounds checking, it is possible that C array references read or write memory outside the allocated bounds of an array. When arrays are allocated on the program stack (a common C idiom) buffer overruns can alter contents of a stack frame, including the return address of a procedure.

Changing return addresses is all that is required to breach the security of standard operating systems, including Linux. The hacker kits now available on the Internet are highly automated and rely heavily on known buffer overruns to crack Internet hosts. In recent years 40%-50% of newly discovered security vulnerabilities have been buffer overrun exploits [138]. In [138], a program analysis system is described for detecting buffer overruns in C programs. The analysis is formulated as an integer constraint resolution problem, where the goal is to determine whether integer variables representing indices into an array are guaranteed to fall inside the bounds of the array.

Because of the ready availability of Open Source, we were able to test this system and measure its performance on hundreds of thousands of lines of C code. The system discovered previously unknown buffer overruns in the Linux `nettools` package and `sendmail` (both of these security-critical programs run in millions of Internet hosts). In addition to the empirical results, new theory and algorithms were developed for the efficient resolution of the particular class of integer constraints arising in this context.

C.1.2 Conceptual Drivers: Specifications and the Interaction among Program Analysis, Model Checking, and Theorem Proving

A theme that cuts across all areas of the project is the potential role of software specifications. While we aim for fully automatic tools that require no interaction with the programmer beyond examination of the source text of the program and simulation or execution of the same, we believe that for some important problems automated solutions are too expensive (i.e., the lower bounds are super-polynomial).

In such situations, it is natural to ask what information the programmer could supply to make the problems tractable. Designing such specification languages is itself challenging, even if the specifications are specialized; the goal is to add enough information to make the analysis feasible without significantly burdening programmers with many new requirements in writing programs.

For those problems where a combination of techniques from the different analysis methodologies is used, there is the question of how work is divided between the multiple analysis algorithms. For example, in model checking concurrent software for, say, deadlocks, it is necessary to work with an abstraction of the original program because working directly with the original source yields is beyond the capability of current model checking technology. This abstraction is normally computed by classical program analysis techniques, which yields a *skeleton* of the program [122, 124] that is small enough to be model checked.

While natural, this architecture suffers from the drawback that results are limited by any imprecision in both components, and in fact imprecisions in the analysis tend to be magnified by any imprecision in the model checker. We will have more to say about this specific example in subsequent sections, but a general theme of the project will be to new approaches to combining program analysis, verification, and model checking that allow the techniques to be scaled readily to large systems.

C.2 The Roles of the PI's

Alex Aiken (UC Berkeley) recently has worked primarily on program analysis, with an eye towards building software engineering tools to support the understanding and evolution of very large software systems.

Tom Henzinger's (UC Berkeley) recent work focuses on compositional approaches to model checking, as well as model checking infinite state systems.

George Necula (UC Berkeley) is an expert in program verification and theorem proving. Prof. Necula's recent work on proof-carrying code and certifying compilation set the stage for the verification work in this proposal.

Dave Schmidt (Kansas State) is one of the few researchers who has worked in both program analysis and model checking. Prof. Schmidt is ideally suited to act as a bridge between the problems that require understanding of both program analysis and model checking techniques.

All four PI's are keenly interested in sharing ideas about the formal analysis of software. It is clear to all the PI's that there are common underpinnings between the different approaches to analysis, although the differences in terminology and emphasis obscure the common foundations. We also believe that integration of the approaches is necessary for future progress, and in particular to allow these techniques to scale smoothly to large systems.

C.3 Initial Focus Areas

We have identified four problem areas where we can initially focus our effort. While some of these problems have been recognized for many years, we feel that recent advances make rapid progress likely. We expect the particular problems we study to evolve throughout the life of the project in response to the progress of the research.

The first area, analysis of device drivers, was chosen because significant progress requires integration of program analysis, model checking, and verification techniques. Furthermore, as we discuss, there are reasons to believe such progress is possible. The next three problem areas are specific supporting technologies that we plan to develop to support the overall goals of the project. A common theme of all four target areas is an investigation of the potential role of programmer specifications.

The focus areas are:

- *Verification of Device Drivers* Device drivers are a constant source of system crashes for millions of users. As we shall argue, device drivers also happen to be well suited to formal treatment via an integration of verification, model checking, and program analysis techniques.

- *Model-checking software* Applying model checking to software requires reexamination of the assumptions of model checking algorithms for hardware.

- *Alias analysis* In recent years the research community has recognized the critical role of alias analysis. Recent progress has been substantial, but the problem is not yet resolved.

- *Memory management* Errors in memory management remain among the most difficult and long lasting sources of software defects.

In the subsections that follow we discuss each of these areas in more detail.

C.3.1 Verification of Device Drivers

Device drivers manage the interaction between an external device (a disk, a CD, an ethernet connection) and the operating system of a computer. Device drivers are essentially embedded systems within a general purpose computing system.

Device drivers have a number of unique properties that make achieving high quality unusually difficult:

- Drivers execute concurrently. As is well known, producing correct concurrent programs is extremely difficult. Testing can rarely uncover all possible problems.

- Drivers are associated with physical devices, and there are many such devices in the world and more are produced all the time. There are also many different versions of operating systems (the Linux kernel has had at least one major release per year for several years). No vendor has the resources to test all possible combinations of drivers and OS releases.

- Drivers are typically written and tested by parties other than those who write the host operating system. Misunderstandings of the interface between the driver and the OS are commonplace; both sides must adhere to a common interface if the complete system is to work.

- Drivers have stringent performance needs, which potentially complicates their implementation.

- Driver safety is critical to the safety of the entire system. As drivers are normally dynamically linked into the operating system, bugs in drivers can crash or lock up the operating system.

Two examples from our own recent experience illustrates these points. Within the last year 34GB disk drives became available in mass production quantities from IBM. However, stock Linux distributions would not even boot after being installed on such drives. The reason: The disk device driver encoded the number of gigabytes available on the drive as a C `short`, a holdover from the days when drives larger than 32GB seemed far in the future.

A particular Linux ethernet driver behaves well under moderate load, but under heavy load—a large transfer that saturates the network for several minutes—throughput drops significantly. The problem is not exactly a performance bug; the driver moves data at full speed for a period of time, but then hangs for several minutes (sending no data) before resuming normal transmission. Despite some effort, the source of this bug was not isolated.

To summarize, like all embedded systems, exhaustive testing is even less feasible for device drivers than it is for other types of software. The result is that drivers tend not to be well-tested and the resulting problems are bewildering to the end user, who is typically neither an operating system nor a device driver expert.

In contrast, there are several aspects of device drivers that make them attractive for research on static analysis techniques.

- Device drivers tend to be relatively small. While our ultimate goal is to apply our methods to large systems, it makes sense to start with small but realistic examples to distill core issues.

- Despite their small size, there is equal opportunity for applying program analysis, verification, and model checking techniques. The strengths and weaknesses of all three approaches are well covered by device drivers (see discussion below).
- A highly stylized, restricted discipline for programming device drivers has emerged over time. For example, a critical correctness condition for any driver is that all executions be finite—control must be returned to the operating system. Inspection of C device driver code shows that loops tend to fall into one of two categories: `for` loops with a clearly bounded number of iterations, or `while` loops with calls to `yield` in the loop body. Thus proving normally intractable properties such as termination is more tractable in the context of device drivers.
- The interface of device drivers to operating systems has been well documented. While still an informal specification, there is at least a widespread consensus on what driver behavior is correct and what is incorrect [119].

Device drivers are linked directly into the operating system; the OS has no boundary of protection from drivers that misbehave. Thus, verification should be aimed at proving that drivers are safe extensions of the host OS. The goal of our work on device drivers is to produce proofs that drivers adhere to the required interface with the operating system. Among the properties that must be verified are memory safety, type safety (in particular, safety of critical internal operating system data structures in the presence of operations by drivers), and termination.

To carry out this work we will really heavily on the techniques of *translation validation* [107] and *proof-carrying code* [106]. Device drivers are invariably distributed as dynamically loaded object files. Thus, any meaningful proofs of correct behavior must be attached to and verifiable from the object code that the end user links into her operating system. Translation validation and proof-carry code provide mechanisms for carrying proofs from the source level down to the object code and then allowing cheap verification of the proof by a client.

Translation validation is a method for proving that a compiler has correctly translated a particular program from the high-level source language into low level assembly language or object code. Compilers bridge between the high-level languages that we use to write software and the machine code that is actually executed. This abstraction gap simplifies programming considerably but it also introduces a gap between the code on which the reasoning and software analyses are performed and the code that is actually distributed and executed. The question of compiler correctness is as old as the first compiler implementations; in a paper published in 1963, John McCarthy refers to this problem as "*one of the most interesting and useful goals for the mathematical science of computation*" [104]. The key distinction between translation validation and work on compiler correctness is that in translation validation we are interested only in the correctness of each compilation, and not in the question of whether the compiler compiles all programs correctly. This considerably simpler problem is nevertheless challenging and translation validation is a current topic of research.

Once a proof of correctness has been produced by translation validation, proof-carrying code (PCC) provides an infrastructure that allows end users to download a device driver with its proof and efficiently verify that the proof shows that the device driver is in fact safe to use prior to linking it into the host operating system. A key benefit of PCC is techniques for *proof optimization*, which compress the representation of proofs dramatically while still preserving the ability to rapidly check the correctness of a proof associated with a driver.

To prove the correctness of device drivers we must do more than prove that a compiler's translation preserves behavior. We do not know a priori that the source code for the driver is correct. Thus we must first use other techniques to construct such a proof for the properties we wish to check, and then carry this proof through the compilation process via translation validation. Because these proofs are likely to have rich logical structure, new work in translation validation is needed to show that the proven properties factor through compilation.

Model checking extended to software is a natural approach to proving that a device driver meets its specification. Many of the requirements imposed on device drivers are that critical events (e.g., the acquisition and release of system resources) must respect certain ordering constraints, regardless of which path is taken through the code or what exceptional conditions arise. Such requirements have the essential characteristics of path reachability problems and should be amenable to model checking techniques. We discuss software model checking in more detail in Section C.3.2.

A particularly difficult problem in any analysis of realistic software is proving static properties about pointers. For example, in the absence of special information about writes through pointers in C, one must assume that any memory location could be affected by the write. Since device drivers manipulate both their own and operating system state through pointer data structures, model checking will need the support of pointer alias information to make significant headway. Here we plan to use our expertise in conservative program analysis, and particular in alias analysis, to support higher level analyses such as model checking. Alias analysis is discussed further in Section C.3.3.

Another area of particular concern is memory management. Low-level C programming encourages the use of explicit allocation and deallocation memory. *Region-based* memory management, an approach used in many systems-level applications, simplifies the reasoning about memory management for both automated tools and the programmer. We plan to explore to what degree structuring memory using regions (see Section C.3.4) can enable the proofs of memory safety for realistic device drivers.

C.3.2 Model-Checking Software

The technique called *model checking* has led in recent years to startling breakthroughs in hardware verification (cf. [39]). A main advantage of model checking, which is based on state-space exploration, is its high degree of automation: unlike theorem proving, which often requires human expertise in the handling of bulky logical formulas, model checking is a "push-button" technology. With model checking, ordinary design engineers, who have expert knowledge about their designs but not about formal methods, can "push a button" to start an algorithm that will check a design for errors. The promise of the technology is such that companies like Intel are building up large in-house research efforts to transition model checking into their design process.

The successful application of model checking often depends on four key characteristics of typical hardware designs. First, the basic structuring mechanism of hardware is *parallel composition*. Hardware is therefore naturally captured by the simple model of a (synchronous or asynchronous) product of state-transition systems. Second, hardware is *static*. Hardware is therefore naturally specified by system-level requirements and verified at design time, once and for all. Third, hardware is often *regular*. Hardware therefore lends itself to certain patterns of abstraction (such as data-path abstraction) and to the discovery of abstractions (such as symmetries). Fourth, hardware is *finite-state*. Hardware therefore permits efficient techniques for state representation, namely, binary decision diagrams. These four characteristics enable the four steps of the traditional model-checking methodology:

1. Flatten a given, complete design into a huge but finite state-transition graph (Kripke Structure).
2. Specify global graph requirements as formulas of a linear or branching temporal logic.
3. Reduce the graph by detecting redundancies that are irrelevant to the requirement specification, for example, by abstracting data paths, or folding symmetries, or partially ordering independent events.
4. Explore the abstract graph using symbolic representations for boolean states such as binary decision diagrams (BDDs).

We call these four-step methodology *closed model checking*, because it views a complete design in isolation, without reference to ongoing design modifications and without interference from other designs, and because it views a particular algorithm, such as CTL model checking, in isolation, without reference to possibilities of adapting and interacting approaches to verification.

In comparison to hardware design, the successful applications of model checking in software have been few and apart (see [40, 42]). The blame is often assigned to the so-called *state-explosion problem*: since model checking is based on exhaustive state-space exploration, the size of the state space of a design is the main limiting factor of the technology. But using symbolic rather than enumerative state representations, extremely large state spaces are routinely explored in hardware verification [34], and even infinite state spaces are explored in timing verification [13, 20]. So it is not strictly the size but the structure of a state space that determines model-checking success or failure. The closed approach to model checking explicitly destroys all structure in Step 1, and then rediscovers some of it in Step 3. We pursue an alternative approach to model checking which exploits rather than destroys design structure.

If we look at the structure of large software systems, we find very different characteristics from hardware designs.

- Besides parallel composition, also *sequential composition* and *instantiation* are fundamental structuring mechanisms of software. This calls into question the adequacy of the state-transition model.
- Software systems *evolve*, as components are modified or upgraded, and missions or requirements change. This dramatically reduces the usefulness of one-time, system-level specification and verification.
- While well-designed software systems do have structure, this structure can be highly *irregular* and individualistic. This dramatically decreases the effectiveness of automatic techniques for the discovery of regularities and abstractions. In fact, most successful applications of model checking to software have obtained abstractions by hand. Fully automated abstractions tend to be close to standard dataflow graphs; while useful, dataflow graphs are not expressive enough for checking most interesting properties [131, 99]. For example, dataflow abstractions do not capture pointer aliasing information (see Section C.3.3).
- Most software is *infinite-state*, as programs operate with data types that range over infinite domains such as the integers. The resulting state spaces cannot be represented efficiently using BDDs. These properties of software may help explain why the potential of model checking for software analysis has yet to be realized. In an attempt to adjust the model-checking technology to the characteristics of software systems, we plan to develop and investigate a methodology based on the following ideas, which we refer to as *open model checking*.

Component modeling and specification Transition-system models extracted from software are often either too expensive for analysis (if the model is too detailed) or too abstract for establishing the desired properties (if the model is not detailed enough). As an extension to transition-system models we investigate *Hierarchical Component Models* (HCMs), where individual components can be left unspecified ("black box"), be specified by a behavioral interface type (see below), be specified by refinement into subcomponents, or be specified, on the lowest level, by a full transition system. Various properties can then be established at the appropriate level of detail. For example, in order to prove an invariant of a certain component, it may not be necessary to know anything about the context of the component, or it may be necessary to know a temporal property about the context, or it may be necessary (in the worst case) to know the full transition system of the context. In hardware verification, this paradigm of reasoning about a component by making abstract assumptions about the context is called *assume-guarantee reasoning*. Traditional assume-guarantee reasoning, however, is confined to the case where the only operation for combining components is parallel composition. Our objective, therefore, is to design an HCM that permits parallel as well as sequential composition of components, and permits sharing as well as instantiation of components. Then, we want to develop a general methodology for assume-guarantee reasoning for such an HCM.

Let us be more specific of how such an HCM might look. Informally, an HCM should capture the structuring aspects of programming languages (and design languages in general) which are generic, such as block structure, hierarchy, sharing and instantiation. One might start with a dag of components, where components on the same level can be composed either sequentially (as modes) or in parallel (as actors). If the dag represents a description of the system at "design time," then a subcomponent which is shared between two components may either be replicated (instantiated) at "run time," or truly shared. For example, the private variables of multiple instances of a class are replicated, whereas static variables are shared. HCMs with only parallel composition are well-understood (see, e.g., [19]), some progress has been made for HCMs with only sequential composition [15], but no compositional verification principles are known for HCMs with both sequential and parallel composition (such as Statecharts [65]), and the situation is even less mature from a formalist point of view for HCMs with instantiation mechanisms (UML [120]).

In assume-guarantee verification, components must be abstracted at various levels of detail. Instead of attempting to discover suitable abstractions from a complete implementation description of a component, we want to provide the software engineer with *Behavioral Interface Types* (BITs) for components. Beyond specifying the data types of parameters and invariant assertions (e.g., that parameters are not aliased) which must hold when a method is called, BITs also permit the specification of temporal requirements. For example, it may be required that the call of one method must always be preceded by the call of another (initialization) method. Or, in a real-time system, it may be required that a certain function is called every p time units. Unlike invariant assertions, which constrain only individual states, BITs may contain constructs from temporal logic which can express such precedence and periodicity constraints. BITs serve similar purposes as documentation, but they can be understood and exploited by automatic verification tools on two levels. First, a BIT can serve as an abstraction of the actual component, and as such can be used to establish higher-level properties, or checked for consistency against the BIT of another component. Second, it can be checked if the actual component satisfies the BIT.

While linear and branching temporal logics are adequate for system-level specification, more expressive BITs can be based on *Alternating Temporal Logic* (ATL) [21]. ATL views the components of a system as independent players in a game, and enables the specification of component-level requirements. For example, in ATL it is possible to specify that a component can attain a goal regardless of the context into which the component is dropped (this is not possible in LTL or CTL). Unlike system-level requirements, component-level requirements can be modified and reverified individually whenever the corresponding components are changed. The model checking of ATL requires the solution of games [45, 44].

Infinite-state exploration While model checking was originally invented for the analysis of finite-state systems, much recent interest has concerned the application of such methods to infinite-state systems (e.g., pushdown systems [31, 35], variants of Petri nets [51], systems with unbounded buffers [1], parametric numbers of processes [32, 61], systems with integer variables [141, 33] or real-numbered clock variables [13], hybrid systems [69]). There are essentially two kinds of approaches. Approaches of the first kind reduce an infinite-state system to an "equivalent" finite-state system, and then explore the resulting finite state space (e.g., the region-graph method for timed automata [13]). We call these approaches *reductionist*. Approaches of the second kind explore the infinite state space directly, by manipulating constraints that may represent infinite state sets (e.g., the clock-zone method for timed automata [87]). We call these approaches *symbolic*. While perhaps optimal in theoretical complexity, reductionist approaches usually experience state explosion, and are typically outperformed in practice by symbolic approaches. Also, in practice it is often difficult to find a suitable reduction of a given system ("abstraction").

In symbolic approaches, nonboolean data types can be handled by separating the two main algorithmic concerns in model checking, namely, global state-space traversal and local state manipulation. For example, reachability can be checked by iterating a successor operator on states. The computation of the successor operator is a *local* concern, which depends on the data types and operations for the variables that make up the state space; the iteration of the successor operator is a *global* concern, with the main issue for infinite-state systems being termination (Under what circumstances is termination guaranteed? [85] If termination is not guaranteed, can the fixpoint be computed in a finite number of steps by "acceleration methods"? [141]). By separating the local from the global concerns, we can investigate a variety of novel combinations of automatic methods (decision procedures, model checking, fixpoint acceleration), interactive methods (theorem proving), and approximation methods (domain or predicate abstraction, fixpoint extrapolation) for either one or both concerns. For example, we may want to compute the successor operator of an integer program automatically using a decision procedure for quantifier elimination, then iterate it a few times in order to obtain an educated guess of what the fixpoint might be, and finally verify the guess using inductive theorem proving [93].

Formally, symbolic algorithms for infinite-state analysis operate on region algebras, where here the term "region" refers to the finite representative of a possibly infinite set of states [85].[1] A *region algebra* consists of a set of regions (say, constraints in some constraint language) which is effectively closed under the predecessor operation *Pre*, the successor operation *Post*, boolean operations, and permits nonemptiness (satisfiability) checking for regions. For example, for the analysis of real-time systems, a suitable set of regions are the boolean combinations of constraints of the form $x + c \leq y + d$, for clocks x and y and integers c and d [87]. A procedure which repeatedly applies the operations of a region algebra may or may not terminate, and following [57], we refer to it as a *symbolic semi-algorithms*. We have classified infinite-state systems into four categories, according to which type of symbolic semi-algorithms terminate [69, 85]. Systems of the first class have finite *bounded-reach equivalence* quotients (two states are bounded-reach equivalent if they can reach the same observables in the same number of transitions). For these systems, which are the *well-structured* transition systems of [2, 57], all semi-algorithms that iterate the *Pre* operation on constraints terminate; this includes, in particular, symbolic safety and reachability analysis. Systems of the second class have

[1] Note that we use region for an entirely different notion in Section C.3.4; unfortunately this name clash just reflects the standard terminology in the literature.

finite *trace equivalence* quotients. For these systems, all semi-algorithms that iterate the *Pre* operation and intersection with observables terminate. This permits the symbolic model checking of all LTL and ω-regular trace properties [86]. Systems of the third class have finite *similarity* quotients, which implies that all semi-algorithms which iterate the *Pre* operation and positive boolean operations on constraints terminate. This permits the symbolic model checking of universal and existential CTL and μ-calculus properties. Finally, systems of the fourth class have finite *bisimilarity* quotients. They are the most restrictive class and ensure the termination of all semi-algorithms that iterate the *Pre* operation and boolean operations, which permits the symbolic model checking of the full CTL and μ-calculus.

Since we have argued to move from system-level behavioral specifications (LTL and CTL) to component-level behavioral specifications (ATL), we need to extend this framework for the symbolic analysis of infinite-state systems from single-player structures (transition systems) to multi-player structures (concurrent games); that is, we need to obtain symbolic semi-algorithms for solving infinite-state games. In each round of an infinite concurrent game, all players simultaneously choose moves according to some deterministic strategies and the choice of moves determines a set of possible next states (games in which the players take turns are a special case). In this way an infinite sequence of states is produced. Player 1 may have as objective to keep an observable invariant (safety), to reach an observable (reachability), or more generally, to satisfy an ω-regular condition on observables. In order to check a winning condition, such as "Player 1 has a strategy to reach an observable no matter how player 2 behaves," we need to iterate a controllable-predecessor ($CPre_1$) operation: for a set σ of states, $CPre_1(\sigma)$ contains those states from which player 1 can force the game into σ in a single round by choosing an appropriate move. By computing on region algebras with $CPre_i$ operations, for all players i, we hope to obtain symbolic semi-algorithms for solving all ω-regular infinite-state games. This would allow us to classify infinite-state systems as to which ATL properties can be model checked.

C.3.3 Alias Analysis

Within the last year it has become possible to apply program analysis techniques to the problem of *alias analysis* to the largest Open Source programs: `gcc`, `gimp`—the GNU Image Manipulation Program—and the Linux kernel. We plan to use this new capability to make a systematic study of alias analysis, which is arguably the most important and difficult of program analysis problems.

Alias analysis is also a paradigmatic example of a large class of analyses that can be naturally expressed using *atomic set constraints*. Besides pointer analyses, this class includes closure analysis for functional languages, receiver class analysis for object-oriented languages, among many others [7, 27, 58, 66, 98, 103, 111, 117, 127]. Constraint-based analyses express desired information about a program as constraints; constraint resolution computes the information. Recent advances in constraint resolution algorithms have increased the size of programs that can be analyzed by orders of magnitude; relatively expressive analyses have been scaled to hundreds of thousands and even millions of lines of code [55, 135, 43].

Aliasing occurs when two program names refer to the same object. Accounting for potential aliasing is a basic problem for compilers and software engineering tools for essentially every widely used programming language (most notably for C, C++, and Java). Consider the following C-like code fragment:

```
*x = 3;
*y = 4;
printf(*x);
```

This program may print 3, but it may also print 4 if `y` and `x` refer to the same memory location. This tiny example illustrates both the importance and difficulty of dealing with aliasing.

Alias analysis is important because potential aliasing is pervasive in programs written in C, C++, and Java. Operations on pointers (in C and C++) and references (in Java) are ubiquitous in these languages. As the example illustrates, any assignment through the pointer `y` leads to a significant loss of static information for every other pointer value,[2] unless we know by some means that `y` does not alias other pointers. Thus every tool that seeks to infer program properties or perform program transformations is dependent on the quality of aliasing information available for the program. The goal of alias analysis is to prove as many

[2] In Java we can trivially confine the loss of information to pointers of a compatible type, but unrestricted casts in C and C++ mean that non-trivial analysis is needed even for pointers of incompatible types.

pair-wise *no-alias* relationships between values as possible; for the remaining *may-alias* relationships aliasing of the two values may or may not exist at runtime.

Alias analysis is difficult because the information needed to prove no-alias relationships need not be confined to a small portion of the program. In the example above, the information about what `y` references is not in the fragment shown; we must look at a larger program context to learn the relevant information for `y`. The trend in the research literature is inexorably towards the analysis of larger and larger portions of programs. The search for alias analysis algorithms that are both scalable to large programs and precise enough to be useful has proven to be a major research challenge and has produced a large literature in recent years [36, 38, 43, 46, 47, 48, 49, 56, 96, 101, 121, 126, 140, 142, 143].

In previous work we focused on scaling program analyses, and in particular alias analysis, to very large programs [55, 135], including all of the major Open Source programs. We plan to use this ability in two ways to advance our understanding of alias analysis.

Dynamic Measures of Quality Ideally the quality of a new program analysis should be characterized as a cost/benefit ratio, where the cost is the time and space required to compute the analysis, and the benefit is the precision of the analysis, at least relative to other analyses that solve the same problem. Historically, there are two approaches to measuring the benefit of alias analyses:

- *Average points-to set size* We compute the average number of static aliases per dereference site over the entire program.
- *Speedup in a compiler* The performance of a test program compiled with an optimizing compiler is measured with and without the static alias analysis. Benefit is an average over a test suite of the speedup obtained through use of the alias analysis.

Both measures suffer from a serious problem. Currently we do not know, for example, whether the algorithms we study detect 5% or 95% of the possible no-alias relationships in programs. Without such measures we cannot know when the alias analysis problem is solved.

Following ideas best expressed by [50], we will build an instrumented test infrastructure to measure how many of the may-alias relationships inferred by alias analyses exist at runtime. In the example above, we would first gather traces of sample program executions and check whether in fact the values of `x` and `y` are ever aliased in the execution trace. The quality of an analysis is the ratio

$$\frac{(\#ofmay-aliasrelationshipsinferredbyanalysis)}{(\#ofactualaliasesmeasuredbytests)}$$

which measures the percentage of likely spurious may-alias relationships computed by the analysis. A ratio of 1 is a perfect analysis; a score of 1.1 indicates that the alias analysis probably overestimates the number of actual aliases by about 10%. The qualification "probably overestimates" is necessary because the number of true aliases detected by analyzing traces depends on the quality of the test suite. For Open Source programs enormous quantities of test data are readily available.

A challenge is scaling trace analysis technology to large programs and long executions for the study of aliasing invariants. While the results reported in [50] are encouraging and large traces have been gathered in other contexts for many years [128], new compression techniques and optimizations may be needed for analyzing dynamic alias information.

Building a successful dynamic testing infrastructure for alias analysis would likely have an impact on other program analyses. Besides gathering information on the accuracy of such analyses, note that the trace analysis can also detect bugs in analyses if facts that are "proven" turn out not to hold at run time. Given the complexity and difficulty of testing modern program analysis techniques, we fully expect to find such bugs.

Language-Level Expression of Alias Information While we consider automatic alias analysis a key building block for any analysis problem involving large software systems, we do not believe it is ultimately sufficient. Deciding whether two program quantities alias one another is an undecidable property which, for scalability reasons, we try to approximate with algorithms that run in time nearly linear in the size of the program. The gap between the inherent complexity of the problem and the complexity of the approximation invariably means that there is no way to guarantee that any particular no-alias relationship can be proven by any practical analysis. There are several situations in which it is very useful to check statically that certain no-alias relationships hold:

- Programmers make implicit assumptions about the nature of alias relationships. For example, the correctness of a device driver may rely on the fact that the driver manipulates the sole references to memory locations recording the internal state of the device.
- Programmers rely on compilers to take full advantage of optimization opportunities in performance-critical sections of code, which in turn may rest on proving certain no-alias relationships. Current practice is that the programmer guesses what the compiler needs to know, restructures the program to make the truth of the intuited facts plain, and recompiles. This process is repeated until either the program runs fast enough or the programmer gives up.
- To aid optimizing compilers, both FORTRAN and C9X have support for asserting that function parameters have no aliases.[3]. These assertions are not checked.
- Failure to prove critical no-alias relationships may cause higher-level static tools, such as model checkers, to fail to prove expected properties.

For all of these reasons, we expect that building reliable large systems can be aided by some form of programmer-supplied, statically checked declarations about aliasing. This gives the programmer confidence that the expected properties are in fact true and removes both the guesswork and subtle errors that arise with the interactions between implicit assumptions, unchecked assertions, and other complex tools. We plan to begin by examining to what degree C9X's `restrict` qualifiers can be inferred, and what additional statements are needed to support such inference. In our experience, an inference-based approach often yields systems that formalize otherwise implicit software properties as part of the programming language with only a small additional burden placed on the programmer [6, 60, 102]. Another starting point is type systems based on linear logic developed in the functional programming community [29]. While primarily designed to guarantee single-threaded computations for the purposes of destructively updating arrays or global state, these systems might be adapted to the problem of static checking of alias relationships.

Specification of Polymorphic Interfaces A more general problem is the specification of polymorphic program analysis properties. Polymorphic signatures arise naturally for any function that may be used in multiple, distinct contexts. One standard methodology for formalizing polymorphic analysis is to use a polymorphic type system with a rich underlying type language. If we allow users to specify the expected outcome of analyses, we must also consider the specification of polymorphic analyses. These specifications must be checked by inference-based tools for consistency.

For constraint-based analyses polymorphic statements have the form $\forall\alpha.\tau/C$, where C is a system of constraints associated with the type τ. Checking polymorphic specifications requires checking that a automatically inferred polymorphic type is at least as general as a programmer-specified polymorphic type. This reduces to a constraint *entailment* problem: Given constraint systems C_1 and C_2, is every solution of C_1 also a solution of C_2?

In comparison to constraint resolution algorithms, progress on constraint entailment has been slow. Flanagan and Felleisen [59] show that an entailment problem for a particular class of set constraints is PSPACE-hard. Henglein and Rehof [67, 68] consider the simpler entailment problem for subtyping constraints. They show that structural subtyping entailment for constraints over simple types is coNP-complete and that for recursive types is PSPACE-complete, and the non-structural entailment for both simple types and recursive types is PSPACE-hard.

Previous work can be seen as moving progressively from richer languages to simpler ones in search of tractable entailment problems. We plan to take the opposite tack: Begin with very simple constraint languages with tractable entailment problems and add expressive power until we find the boundary between tractable and intractable entailment problems. For example, for standard ML-style polymorphic types we have found an efficient entailment algorithm based on congruence closure; the situation for extensions with constraints is yet to be explored. Our overall goal is characterize constraint languages with tractable

[3] In FORTRAN the language definition requires that function parameters not be aliased. C9X has added a type qualifier `restrict` to the language [37]

entailment problems; these are the languages most suitable as a basis for specification of polymorphic analysis properties.

C.3.4 Memory Management

The most widely used form of memory management is explicit allocation and deallocation, as in C's malloc/free C is very widely used for system programming tasks (the Linux kernel, for example, has about 550,000 lines of C code). Malloc/free style programming requires programmers to explicitly keep track of when memory is safe to release. Freeing memory too early (resulting in dangling pointers) or too late (resulting in memory leaks) are some two of the most pernicious and difficult to remove sources of bugs in C programs.

A well-known alternative to malloc/free memory management is garbage collection [139]. Garbage collection has come into the mainstream with Java [62] and is now widely used for tasks that are not performance critical. Despite the availability of conservative garbage collectors for C [30], garbage collection is not widely used for performance-oriented systems programming, most likely because garbage collection requires the programmer to give up control of memory management, with a corresponding loss of precise control over system performance and resource utilization. In the case of embedded systems such as device drivers, unpredictable pauses (even short ones) are also problematic.

A third approach to memory management is *region-based* memory allocation, which has been widely used as an implementation technique for decades under a variety of names, e.g., zones [118], groups [97], or arenas [64]. Systems as important and diverse as Informix's database server products and the Open Source lcc compiler and Apache WWW server use regions to manage memory. Regions have also recently attracted research attention as a target for static inference of memory management [136] and for improving locality of dynamically allocated data [133].

In a region-based memory allocation scheme each allocated *object* is placed in a program-specified *region*. Memory is reclaimed by destroying a region, freeing all the objects allocated therein. A simple example is shown below. Each iteration of the loop allocates a small array. The call to `deleteregion` frees all arrays.

```
void f()
{
  Region r = newregion();

  for (i = 0; i < 10; i++) {
    int *x = ralloc(r, (i + 1) * sizeof(int));
    work(i, x);
  }
  deleteregion(r);
}
```

In a recent paper [60], we showed that region-based programming (where programmers explicitly allocate objects in regions) is more efficient than malloc/free in C programs, both in time and space consumed. For example, allocation is about twice as fast and deallocation is much faster. We also found that explicit regions have some particular strengths. First, regions bring structure to memory management, making programs clearer and, in our subjective assessment, easier to write compared to using malloc/free. For example, it is not necessary to walk through a complex data structure to deallocate it. Others have appreciated the structuring advantages of regions before; a portion of the Apache documentation reads (regions are called *resource pools* in Apache terminology):

> ...a `pool` is a pointer to a resource pool structure; these are used by the server to keep track of the memory which has been allocated, files opened, etc., either to service a particular request, or to handle the process of configuring itself. That way, when the request is over (or, for the configuration pool, when the server is restarting), the memory can be freed, and the files closed, en masse, without anyone having to write explicit code to track them all down and dispose of them [28]

Second, we found some evidence that regions can be used to provide significantly improved data locality, as posited by Stoutamire [133]. The execution time of one benchmark was improved 24% simply by reorganizing allocation so that the most frequently accessed objects are allocated in a single region. Neither malloc/free nor garbage-collected systems provide any mechanism for expressing locality. Third, regions are useful for building software with predictable performance, as the cost of operations is easily bounded.

In [60] we also showed initial results that regions could be made safe through a combination of run-time tests and compile-time analysis. While this adds some performance overhead (about 20% in our initial experiments) we believe this overhead can be made small, and it suggests that adding regions as a feature in a systems programming language (such as C) would allow one to write programs that are both more efficient, predictable, reliable, and easier to maintain than existing systems programs.

We illustrate our approach to adding language support for safe regions with one technical idea; we omit others due to space constraints. When a region r is freed, a safe implementation must in some way keep track of any pointers from other regions to region r. Such *cross-region* pointers are potentially left dangling when r is freed. From studies of region-based programs, we know that the vast majority of pointers are *same-region* (i.e., not cross-region). Having the programmer declare same-region pointers would aid static checking and reduce run-time tests, since same-region pointers need not even be inspected when a region is freed.

Consider a typical linked-list C data structure:

```
struct list {
    char data *x;
    struct list *next;
};
struct data {
    int f1, f2;
};
```

Typically all elements of the list have the same lifetime, so it is natural to allocate all elements in a single region. Assume for the sake of this example that the `data` elements are longer lived than the list. Using `!` to denote same-region pointers and $*$ to denote any pointer (same- or cross-region), a logical declaration is:

```
struct list {
    char data *x;
    struct list !next;
};
struct data {
    int f1, f2;
};
```

Internal to the compiler, existential types can be used to formalize the same-region relationship. The type

$$list[\sigma] = \{x : *\exists\sigma'.data[\sigma']; next : *list[\sigma];\}\ at\ \sigma;$$

captures the intuition that for a `list` structure allocated in region σ, it's `next` field points into region σ as well, but it's `x` field points into some statically unknown region (we merely know that it exists).

We expect that an existential region system can be extended to all of C and be type checked using variations on the standard methods for treating existential types. The key issues are the subtle tradeoffs between how much specification the programmer is expected to give about use of regions and the balance between what is checked statically and dynamically.

C.4 Research Infrastructure

The project would begin with immediate access to existing tools for and benchmark programs drawn from Open Source software. Prof. Aiken's previous work on BANE (the Berkeley ANalysis Engine) has produced an ANSI C compatible front end for C (parsing, semantic analysis, and type checking) and the GNU extensions to C. This benchmark suite includes over 60 Open Source C programs, including all of the largest ones (`gcc`, `gimp`, `postgres`, `gimp`, etc.). Prof. Necula's preliminary work on translation validation has also been done using `gcc`.

From Prof. Henzinger's previous work the project also has access to the state-of-the art model checking tools HyTech and Mocha. There is also considerable expertise within the group in other well-known model checking tools and theorem-proving systems.

The project also has access to Java front-ends, including a byte code parser and analyzer developed as part of BANE and a Java 1.1 front-end from Kansas State University. Acquisition of a Java 1.2 front end will be needed to conduct work on many Open Source Java programs, but note that many commercial compilers do not currently handle all of the Java 1.2 specification. We expect that the frequent changes to the Java language definition will substantially diminish over the next 2 years, making it feasible to maintain a Java research infrastructure for the longer term.

C.5 Budget Justification

For the three co-PI's at Berkeley, the budget includes funds to support graduate students, summer salary for the faculty, and funds for desktop equipment, travel, and administrative support. For Prof. Schmidt, who is based at Kansas State University, the budget provides additional travel money to allow Prof. Schmidt to visit Berkeley on a regular basis to carry on technical discussions and participate directly in the work on analysis and model checking.

A large single item in the budget is money for 10 high performance PC's, to be used as a compute cluster for experimentation. We expect many of the algorithms we investigate to be computationally intensive, particularly when dealing with any of the formal analyses of very large programs. While computationally intensive, we also expect many of our algorithms to be highly parallelizable at a coarse granularity (e.g., the trace collection work for studying alias analysis, and state space exploration for large model checking problems). A cluster of powerful PC's is the most direct and cost effective way to experiment with such algorithms.

C.6 Results from Previous NSF Support

Alex Aiken has been involved in two separate NSF-funded research efforts in the last several years. The most closely related project is BANE (for the The Berkeley ANalysis Engine). The PI was the sole principal investigator on two related grants that supported this work. The two awards are:

- National Young Investigator Award, NSF Grant No. CCR-9457812, $268,021, 9/01/95 – 8/31/00 "Constraint-Based Program Analysis", PI: Alexander Aiken
- NSF Grant No. CCR-9416973, $134,821, 9/01/95 – 8/31/98 "Program Analysis: Logics, Algorithms, and Applications", PI: Alexander Aiken

The second grant functioned effectively as a supplement of the NYI.

The BANE project focuses on the use of constraint-based program analysis techniques to improve software development processes. Constraints have many advantages as a paradigm for constructing large program analysis systems, and it is fair to say that in the last five years constraint-based analysis has become part of the standard toolkit of the software analysis community.

Briefly, constraint-based analyses are divisible into a constraint generation phase that captures relationships between program quantities as local constraints, and a constraint resolution phase that solves the constraints to compute the desired global information about the program. Constraints are a natural conceptual and engineering framework for program analysis. From the viewpoint of the analysis designer, the problem of implementing an analysis is reduced to the problem of writing code to generate appropriate constraints, which is often done with a simple recursive walk of the abstract syntax tree. The analysis engine takes care of all details of constraint representation and resolution. From an engineering perspective, there are many opportunities for optimizing the representation and resolution of constraints that can be encapsulated inside the resolution engine.

The primary goals of the project have been:

1. To develop scalable approaches for implementing constraint-based analyses.

2. To implement a general constraint-solving package using the ideas of (1).

3. To experimentally demonstrate the utility and scalability for a number of distinct applications.

4. To demonstrate usefulness of the system for rapid prototyping of program analyses. In other words, to show that initial implementations of analyses can be constructed much more quickly and cheaply using a general constraint-based framework.

All of these project goals have been achieved. As of 1998, the development of optimized constraint resolution procedures allowed the analysis of 100,000 line programs, which was roughly 10 times the size of any comparable system reported in the literature at that time. Since then, further algorithmic discoveries have increased the analysis capability to 450,000 line programs, which is nearly the largest program available in the public domain.

As part of the project several applications have been developed. Perhaps the most successful of these was a race-detection algorithm for the factory control language RLL. Relay races are common, but very difficult to detect, class of bugs in factory control applications. Our RLL analysis was very successful at finding these bugs in very large, production RLL programs, including one known bug that had originally required four hours of factory down time to repair. This paper was perhaps the first to report the successful application of program analysis technology to detecting bugs in large, production software systems. In addition, the work to develop the prototype analysis was done in BANE in two months, demonstrating the benefits of BANE-like frameworks for rapid prototyping. A paper on this work received the Best Paper Award from the European Association of Programming Languages and Systems federated conference.

One of the unexpected innovations in BANE was the discovery of *mixed constraints*. Mixed constraints provide a very general and useful way to combine multiple constraint formalisms in a single application. The notion of mixed constraints has significant benefits. First, it allows an easily extensible system architecture—a new formalism can be added without perturbing the entire system. Second, users can combine the various constraint languages in whatever way appears most fruitful for a particular task. We found this ability to write *ad hoc* customized constraint languages from general components particularly interesting. The use of mixed constraints allowed us to rapidly prototype applications in a few days that would previously have taken us months to construct.

A distribution of BANE was released in 1998 and has been used by other program analysis researchers world-wide. The distribution and other information about BANE can be obtained from the Web site `http://bane.cs.berkeley.edu`.

The BANE project supported 4 Ph.D. students at various stages of their graduate careers. One of these, now graduated with a Ph.D., has gone on to become a research scientist at Microsoft Research. The other 3 are still in the Ph.D. program at Berkeley and will finish their degrees in approximately one year.

Besides the software distribution, the BANE project produced a number of journal and conference papers [55, 53, 52, 6, 60, 134, 3, 9, 8, 54, 4, 5].

Thomas Henzinger was the PI on a National Science Foundation CAREER award, (Grant CCR-9501708, "Computer-aided verification", 8/1/1995–7/31/1999, $ 133,332). This work developed new models and algorithms for the automatic verification of heterogeneous systems. The main focus was the discrete-continuous heterogeneity of hybrid systems, but other sources of heterogeneity, such as hardware-software heterogeneity and synchronous-asynchronous heterogeneity, were also investigated. In each case, model-checking problems were rendered tractable by developing compositional methods for specification and analysis.

Specifically, a comprehensive mathematical theory for model checking was developed for real-time and hybrid systems. The basic ingredients of the methodology are real-time temporal logics for specification [14, 92, 116, 70], clock-based models of computation [18, 63, 17], efficient algorithms for the analysis of timed systems [83, 11, 80], finite quotient constructions for infinite-state hybrid systems [69, 76, 79], model checking with polyhedral constraints [20, 26] controller synthesis for hybrid systems [77, 75, 78], and approximation techniques for nonlinear dynamics [94, 93, 74, 115]. The methodology has been implemented in the software tool HyTech [71, 72, 73]. HyTech is the first model checker for hybrid systems and has attracted several hundred registered users worldwide; it has been used as analysis tool in numerous scientific publications, for example, in aerospace, automotive, manufacturing, and robotics applications [95, 41, 129, 100, 137].

This work also developed a new model of concurrent computation which supports the modeling of mixed hardware-software and mixed synchronous-asynchronous systems [16, 19]. New techniques were developed for fighting the state-explosion problem in model checking [10, 82, 25, 81], putting special emphasis on assume-guarantee methods for decomposing model-checking tasks [88, 91, 89, 12] and on game-theoretic methods for specifying and verifying adversarial as well as collaborative properties of individual processes [21, 23, 45, 22]. The compositional methods have been implemented in the model-checking environment Mocha [24] and applied to verification problems with mixed hardware-software aspects [90]. Mocha has been successful in very large case studies, such as the verification of a video-graphics-image dataflow chip with 96 processors and 6 million transistors [84].

George Necula has been a PI and co-PI on two NSF grants in the past five years. The primary one, and most related to the proposed effort, is the NSF Career grant CCR-9875171 entitled "CAREER: A Logic-Based Approach to Software System Integrity and Security" that provides one month of summer salary and support for one graduate research assistant (Shree Rahul) for a period of four years starting January 1, 1999, in the total amount of $205,000.

The first year of the project has focused on one of the proposed research directions, namely the study of algorithms for compressing the proofs that accompany the untrusted code in the proof-carrying code paradigm. Proof-carrying code (PCC) [106] is a technique that allows for quick and simple load-time checking of properties of untrusted software by requiring the code producer to package the code together with a proof of its adherence to a set of safety rules. Before the start of the project the typical size of proofs for type safety of optimized machine code was about 250% of the code size. Progress in the direction of proof compression is important because the size of proofs has direct relevance to the practicality of PCC. But care must be taken not to sacrifice the simplicity of the proof checker by requiring the addition of a sophisticated decompression stage.

An early outcome of the project has been the design and implementation of a 3-stage proof-compression pipeline. The first stage exploits the fact that some of the structure of the proof can be reconstructed from the control-flow structure of the program being checked. Thus structural information need no longer be transmitted as part of the proof. The second compressing stage exploits the fact that in typical agents certain proof procedures are reused. This is especially true of those agents that are produced and certified using automatic tools. This property is exploited by allowing untrusted code producers to dynamically extend the database of trusted axioms with proved lemmas resident at the code consumer, which allows the untrusted party to reduce the size of the proofs by rewriting them to make use of the lemmas. These two compression techniques achieve a cumulative compression factor of 3:1. The final compression technique uses Huffman coding to further reduce the size of proofs by another factor of 2. The result is that proofs are now only 25%-50% of the size of the machine code.

David Schmidt was a PI on the recently concluded project, NSF/DARPA CCR-9633388, "High-Assurance Software Evolution," September 1996–February 2000, $200,000. This project investigated the application of static analysis techniques to validating whole programs and ensuring consistency of program modules as they evolve. Schmidt's efforts were directed towards applying abstract interpretation techniques to a program and its operational semantics definition so that a "skeleton" might be extracted for static analysis [122, 124]. The program skeletons produced by this methodology proved especially useful for analysis by means of model checking, and this triggered a reexamination and reformulation of Steffen's seminal efforts [130] at representing data-flow analyses as model checks [123]. Schmidt and Steffen have since joined forces and are jointly examining the applicability and ramifications of model-checking-based implementations of general static analyses [105, 125, 132].

D References

[1] P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127:91–101, 1996.

[2] P. A. Abdulla, K. Čerāns, B. Jonsson, and Yih-Kuan Tsay. General decidability theorems for infinite-state systems. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 313–321. IEEE Computer Society Press, 1996.

[3] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(1999):79–111, 1999.

[4] A. Aiken, M. Faehndrich, J. Foster, and Z. Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the second International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96, Kyoto, Japan, March 1998. Springer-Verlag.

[5] A. Aiken, M. Faehndrich, and Z. Su. Detecting races in Relay Ladder Logic programs. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 184–200, April 1998.

[6] A. Aiken and D. Gay. Barrier Inference. In POPL'98 [113], pages 342–354.

[7] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[8] A. Aiken, E. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping (extended abstract). In *Theoretical Aspects of Computer Software (TACS)*, September 1997.

[9] A. Aiken, E. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. *Higher-Order and Symbolic Computation*, 12(3), October 1999.

[10] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state-space exploration. In O. Grumberg, editor, *CAV 97: Computer-aided Verification*, Lecture Notes in Computer Science 1254, pages 340–351. Springer-Verlag, 1997.

[11] R. Alur, C. Courcoubetis, and T.A. Henzinger. Computing accumulated delays in real-time systems. *Formal Methods in System Design*, 11(2):137–156, 1997.

[12] R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Automating modular verification. In J.C.M. Baeten and S. Mauw, editors, *CONCUR 99: Concurrency Theory*, Lecture Notes in Computer Science 1664, pages 82–97. Springer-Verlag, 1999.

[13] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[14] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[15] R. Alur and R. Grosu. Modular refinement of hierarchic state machines. In *Proceedings of the Annual Symposium on Principles of Programming Languages*. ACM Press, 2000.

[16] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[17] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR 97: Concurrency Theory*, Lecture Notes in Computer Science 1243, pages 74–88. Springer-Verlag, 1997.

[18] R. Alur and T.A. Henzinger. Real-time system = discrete system + clock variables. *Software Tools for Technology Transfer*, 1:86–109, 1997.

[19] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[20] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[21] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.

[22] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference*, Lecture Notes in Computer Science 1536, pages 23–60. Springer-Verlag, 1999.

[23] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.

[24] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.

[25] R. Alur, T.A. Henzinger, and S. Rajamani. Symbolic exploration of transition hierarchies. In B. Steffen, editor, *TACAS 98: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1384, pages 330–344. Springer-Verlag, 1998.

[26] R. Alur, T.A. Henzinger, and H. Wong-Toi. Symbolic analysis of hybrid systems. In *Proceedings of the 36th International Conference on Decision and Control*, pages 702–707. IEEE Press, 1997.

[27] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[28] Shambhala API notes. http://dev.apache.org/API.html#basics.

[29] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, pages 579–612, 1996.

[30] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[31] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: application to model checking. In *CONCUR 97: Concurrency Theory*, Lecture Notes in Computer Science 1243, pages 135–150. Springer-Verlag, 1994.

[32] M.C. Browne, E.C. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.

[33] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite systems using presburger arithmetic. In *CAV 97: Computer-Aided Verification*, Lecture Notes in Computer Science 1254, pages 400–411. Springer-Verlag, 1997.

[34] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[35] O. Burkart and B. Steffen. Pushdown processes: parallel composition and model checking. In *CONCUR 94: Concurrency Theory*, Lecture Notes in Computer Science 836, pages 98–113. Springer-Verlag, 1994.

[36] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer-Verlag, 1994.

[37] C language standard. http://anubis.dkuug.dk/jtc1/sc22/wg14/.

[38] R. Chatterjee, B. Ryder, and W. Landi. Relevant Context Inference. In POPL'99 [114], pages 133–146.

[39] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[40] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 1996.

[41] J.C. Corbett. Timing analysis of ADA tasking programs. *IEEE Transactions on Software Engineering*, 22(7):461–483, 1996.

[42] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM TOPLAS*, 19:253–291, 1997.

[43] M. Das. Unification-based Pointer Analysis with Directional Assignments. In PLDI'00 [108]. To appear.

[44] L. de Alfaro and T.A. Henzinger. Concurrent ω-regular games. In *Proceedings of the 15th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2000.

[45] L. de Alfaro, T.A. Henzinger, and O. Kupferman. Concurrent reachability games. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 564–575. IEEE Computer Society Press, 1998.

[46] S. Debray, R. Muth, and M. Weippert. Alias Analysis of Executable Code. In POPL'98 [113], pages 12–24.

[47] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In PLDI'94 [109], pages 230–241.

[48] N. Dor, M. Rodeh, and M. Sagiv. Detecting Memory Errors via Static Pointer Analysis. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 27–34, Montreal, Canada, June 1998.

[49] M. Emami, R. Ghiya, and L. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In PLDI'94 [109], pages 242–256.

[50] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *The 22nd International Conference on Software Engineering (ICSE 2000)*, page to appear, June 2000.

[51] J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[52] M. Faehndrich and A. Aiken. Making Set-Constraint Based Program Analyses Scale. In *First Workshop on Set Constraints at CP'96*, August 1996. Available as CSD-TR-96-917, University of California at Berkeley.

[53] M. Faehndrich and A. Aiken. Program Analysis using Mixed Term and Set Constraints. In Pascal Van Hentenryck, editor, *Static Analysis, Fourth International Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 114–126, Paris, France, September 1997. Springer-Verlag.

[54] M. Faehndrich and A. Aiken. Refined type inference for ml. In *Proceedings of the 1st Workshop on Types in Compilation*, June 1997.

[55] M. Faehndrich, J. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In PLDI'98 [110], pages 85–96.

[56] M. Faehndrich, J. Rehof, and M. Das. Scalable Context-Sensitive Flow Analysis Using Instantiation Constraints. In PLDI'00 [108]. To appear.

[57] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere. Technical Report LSV-98-4, Laboratoire Spécification et Vérification, 1998.

[58] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[59] Cormac Flanagan and Matthias Felleisen. Componential Set-Based Analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, Las Vegas, Nevada, June 1997.

[60] D. Gay and A. Aiken. Memory Management with Explicit Regions. In PLDI'98 [110], pages 313–323.

[61] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.

[62] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[63] V. Gupta, T.A. Henzinger, and R. Jagadeesan. Robust timed automata. In O. Maler, editor, *HART 97: Hybrid and Real-time Systems*, Lecture Notes in Computer Science 1201, pages 331–345. Springer-Verlag, 1997.

[64] D. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[65] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[66] N. Heintze. Set Based Analysis of ML Programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–17, June 1994.

[67] F. Henglein and J. Rehof. The Complexity of Subtype Entailment for Simple Types. In *Symposium on Logic in Computer Science*, pages 352–361, 1997.

[68] F. Henglein and J. Rehof. Constraint Automata and the Complexity of Recursive Subtype Entailment. In *ICALP98*, pages 616–627, 1998.

[69] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[70] T.A. Henzinger. It's about time: real-time logics reviewed. In D. Sangiorgi and R. de Simone, editors, *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 439–454. Springer-Verlag, 1998.

[71] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 56–65. IEEE Computer Society Press, 1995.

[72] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. In O. Grumberg, editor, *CAV 97: Computer-aided Verification*, Lecture Notes in Computer Science 1254, pages 460–463. Springer-Verlag, 1997.

[73] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[74] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):540–554, 1998.

[75] T.A. Henzinger, B. Horowitz, and R. Majumdar. Rectangular hybrid games. In J.C.M. Baeten and S. Mauw, editors, *CONCUR 99: Concurrency Theory*, Lecture Notes in Computer Science 1664, pages 320–335. Springer-Verlag, 1999.

[76] T.A. Henzinger and P.W. Kopke. State equivalences for rectangular hybrid automata. In U. Montanari and V. Sassone, editors, *CONCUR 96: Concurrency Theory*, Lecture Notes in Computer Science 1119, pages 530–545. Springer-Verlag, 1996.

[77] T.A. Henzinger and P.W. Kopke. Discrete-time control for rectangular hybrid automata. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP 97: Automata, Languages, and Programming*, Lecture Notes in Computer Science 1256, pages 582–593. Springer-Verlag, 1997.

[78] T.A. Henzinger and P.W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221:369–392, 1999.

[79] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[80] T.A. Henzinger and O. Kupferman. From quantity to quality. In O. Maler, editor, *HART 97: Hybrid and Real-time Systems*, Lecture Notes in Computer Science 1201, pages 48–62. Springer-Verlag, 1997.

[81] T.A. Henzinger, O. Kupferman, and S. Qadeer. From *pre*historic to *post*modern symbolic model checking. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 195–206. Springer-Verlag, 1998.

[82] T.A. Henzinger, O. Kupferman, and S.K. Rajamani. Fair simulation. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR 97: Concurrency Theory*, Lecture Notes in Computer Science 1243, pages 273–287. Springer-Verlag, 1997.

[83] T.A. Henzinger, O. Kupferman, and M.Y. Vardi. A space-efficient on-the-fly algorithm for real-time model checking. In U. Montanari and V. Sassone, editors, *CONCUR 96: Concurrency Theory*, Lecture Notes in Computer Science 1119, pages 514–529. Springer-Verlag, 1996.

[84] T.A. Henzinger, X. Liu, S. Qadeer, and S.K. Rajamani. Formal specification and verification of a dataflow processor array. In *Proceedings of the International Conference on Computer-aided Design*, pages 494–499. IEEE Computer Society Press, 1999.

[85] T.A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In H. Reichel and S. Tison, editors, *STACS 00: Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1770, pages 13–34. Springer-Verlag, 2000.

[86] T.A. Henzinger and R. Majumdar. Symbolic model checking for rectangular hybrid systems. In S. Graf, editor, *TACAS 00: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[87] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[88] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 440–451. Springer-Verlag, 1998.

[89] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Assume-guarantee refinement between different time scales. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer-aided Verification*, Lecture Notes in Computer Science 1633, pages 208–221. Springer-Verlag, 1999.

[90] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer-aided Verification*, Lecture Notes in Computer Science 1633, pages 301–315. Springer-Verlag, 1999.

[91] T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-aided Design*, Lecture Notes in Computer Science 1522, pages 421–432. Springer-Verlag, 1998.

[92] T.A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *ICALP 97: Automata, Languages, and Programming*, Lecture Notes in Computer Science 1443, pages 580–591. Springer-Verlag, 1998.

[93] T.A. Henzinger and V. Rusu. Reachability verification for hybrid automata. In T.A. Henzinger and S. Sastry, editors, *HSCC 98: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 1386, pages 190–204. Springer-Verlag, 1998.

[94] T.A. Henzinger and H. Wong-Toi. Linear phase-portrait approximations for nonlinear hybrid systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 377–388. Springer-Verlag, 1996.

[95] T.A. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science 1165, pages 265–282. Springer-Verlag, 1996.

[96] M. Hind and A. Pioli. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In Giorgio Levi, editor, *Static Analysis, Fifth International Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 57–81, Pisa, Italy, September 1998. Springer-Verlag.

[97] Y. Ichisugi and A. Yonezawa. Distributed garbage collection using group reference counting. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.

[98] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[99] M. Klein, D. Koschuetzki, J. Knoop, and B. Steffen. DFA&OPT-MetaFrame: a tool kit for program analysis and optimization. In *Proc. TACAS'96*, pages 422–426. Lecture Notes in Computer Science 1055, Springer, Berlin, 1996.

[100] J. Kosecka, C.J. Tomlin, G. Pappas, and S. Sastry. Generation of conflict resolution manoeuvres for air traffic management. In *IROS 97: Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems*, volume 3, pages 1598–1603. IEEE Press, 1997.

[101] W. Landi and B. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, California, June 1992.

[102] B. Liblit and A. Aiken. Type Systems for Distributed Data Structures. In POPL'00 [112].

[103] S. Marlow and P. Wadler. A Practical Subtyping System For Erlang. In *Proceedings of the International Conference on Functional Programming (ICFP '97)*, pages 136–149, June 1997.

[104] J. McCarthy. Towards a mathematical theory of computation. In Cicely M. Popplewell, editor, *Proceedings of the International Congress on Information Processing*, pages 21–28. North-Holland, 1963.

[105] M. Müller-Olm, D.A. Schmidt, and B. Steffen. Model checking: A tutorial introduction. In G. Filé and A. Cortesi, editors, *Proc. 6th Static Analysis Symposium*. Springer LNCS, 1999.

[106] G. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.

[107] G. Necula. Translation validation for an optimizing compiler. To appear in Programming Language Design and Implementation, PLDI'00, June 2000.

[108] *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver B.C., Canada, June 2000.

[109] *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

[110] *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[111] A. Pnueli and Shalev. What is in a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software*, pages 244–264, 1991. LNCS 526.

[112] *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, January 2000.

[113] *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, January 1998.

[114] *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999.

[115] J. Preußig, S. Kowalewski, H. Wong-Toi, and T.A. Henzinger. An algorithm for the approximative analysis of rectangular automata. In A.P. Ravn and H. Rischel, editors, *FTRTFT 98: Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science 1486, pages 228–240. Springer-Verlag, 1998.

[116] J.-F. Raskin, P.-Y. Schobbens, and T.A. Henzinger. Axioms for real-time logics. In D. Sangiorgi and R. de Simone, editors, *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 219–236. Springer-Verlag, 1998.

[117] J. C. Reynolds. *Automatic Computation of Data Set Definitions*, pages 456–461. Information Processing 68. North-Holland, 1969.

[118] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, August 1967.

[119] A. Rubini. *Linux Device Drivers*. O'Reilly, 1998.

[120] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[121] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In POPL'99 [114], pages 105–118.

[122] D.A. Schmidt. Abstract interpretation of small-step semantics. In M. Dam and F. Orava, editors, *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[123] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, 1998.

[124] D.A. Schmidt. Trace-based abstract interpretation of operational semantics. *J. Lisp and Symbolic Computation*, 10:237–271, 1998.

[125] D.A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In G. Levi, editor, *Proc. 5th Static Analysis Symposium*. Springer LNCS 1503, 1998.

[126] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.

[127] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[128] A. Srivastava and A. Eustace. ATOM—a system for building customized program analysis tools. In PLDI'94 [109], pages 196–205.

[129] T. Stauner, O. Müller, and M. Fuchs. Using HyTech to verify an automotive control system. In O. Maler, editor, *HART 97: Hybrid and Real-time Systems*, Lecture Notes in Computer Science 1201, pages 139–153. Springer-Verlag, 1997.

[130] B. Steffen. Generating data-flow analysis algorithms for modal specifications. *Science of Computer Programming*, 21:115–139, 1993.

[131] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint analysis machine. In I. Lee and S. Smolka, editors, *Proc. CONCUR'95*, volume 962, pages 72–87, 1995.

[132] B. Steffen and D.A. Schmidt, editors. *Handbook of Formal Methods in Program Analysis.* Springer, Berlin, 2001.

[133] D. Stoutamire. *Portable, Modular Expression of Locality.* PhD thesis, University of California at Berkeley, 1997.

[134] Z. Su, M. Faehndrich, and A. Aiken. Detecting races in relay ladder logic programs. *Software Tools for Technology Transfer*, 2000. to appear.

[135] Z. Su, M. Faehndrich, and A. Aiken. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In POPL'00 [112].

[136] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[137] T. Villa, H. Wong-Toi, A. Balluchi, J. Preußig, A. Sangiovanni-Vincentelli, and Y. Watanabe. Formal verification of an automotive engine controller in cutoff mode. In *Proceedings of the 37th International Conference on Decision and Control*, pages 4271–4276. IEEE Press, 1998.

[138] D. Wagner, E. Brewer, J. Foster, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 2000 Network and Distributed Systems Security Conference*, 2000. to appear.

[139] P. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, September 1992. Springer-Verlag.

[140] R. Wilson and M. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 1995.

[141] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 88–97. Springer-Verlag. 1998.

[142] S. Yong, S. Horwitz, and T. Reps. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, Atlanta, Georgia, May 1999.

[143] S. Zhang, B. Ryder, and W. Landi. Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses. In *Fourth Symposium on the Foundations of Software Engineering*, October 1996.