A Portable Compiler-Integrated Approach to Permanent Checking

Nic Volanschi
*my***gcc**
nic.volanschi@free.fr

Abstract

Program checking technology is now a mature technology, but is not yet used on a large scale. We identify one cause of this gap in the decoupling of checking tools from the everyday development tools. To radically change the situation, we explore the integration of simple user-defined checks into the core of every development process: the compiler. The checks we implement express constrained reachability queries in the control flow graph taking the form "from x *to* y *avoiding* z*", where* x*,* y*, and* z *are native code patterns containing a blend of syntactic, semantic and dataflow information. Compiler integration enables continuous checking throughout development, but also a pervasive propagation of checking technology. This integration poses some interesting challenges, but opens up new perspectives. Factorizing analyses between checking and compiling improves both the efficiency and the expressiveness of the checks. Minimalist user properties and language-independent code pattern matching ensure that our approach can be integrated almost for free in any compiler for any language. We illustrate this approach with a full-fledged checking compiler for C. We demonstrate the need for permanent checking by partially analyzing two different releases of the Linux kernel.*

1. Introduction

Checking programs with respect to user-specified properties is an important aspect of automated software engineering. Recent years have seen many advances in software checking materialized in the apparition of many different checking tools performing various levels of checks, ranging from purely syntax checks [7, 29, 1, 20, 13, 11], going through lightweight model checking [18, 28, 10, 14, 3, 8, 24, 19] up to sound software model checking [12, 4, 21]. Despite this apparently very encouraging trend, the use of these tools in everyday software practice is still marginal. Face to this situation, a legitimate question is: why?

There are several possible reasons of this discrepancy, among which:

- efficiency: most of the tools are not fast enough for everyday use. This is especially the case for more formal verifiers.
- usability: the learning curve might be too long for some tools; some other tools are not sufficiently easy to use even for trained users
- integration: most of these tools are not integrated with familiar development tools

As a consequence of these and maybe other reasons, checking tools are not used at all in most software projects. At best, checking is performed sporadically. However, occasional checking has two major shortcomings. Firstly, errors are not instantly caught, meaning that they may be found in later stages when fixing them is more expensive. Secondly, errors that were fixed in some phase of the project may be re-introduced.

This paper explores a pragmatic approach to code checking aiming to incorporate some minimal amount of checking *throughout* the development process. Our approach to permanent checking is based on extending the central tool of the development process, the compiler, with user-defined properties that are checked in addition to compilation.

Achieving integration of user-defined checks within a compiler requires a new balance between power and precision on one hand versus speed and usability on the other hand. We propose here to favor speed and usability based on a minimalist class of user-defined properties, trivial to define by any programmer and checkable very efficiently, yet covering many well-known checks on systems code. The checks we propose are simply expressed as reachability queries in the control flow graph, constrained by syntactic, semantic, and dataflow information.

Our approach aims specifically to be easy to integrate in existing compilers, in order to allow a really widespread use of checking technology by every programmer. To achieve this portability goal, we use a language-independent pattern matching technique that can be implemented very concisely in virtually any compiler. We practically demonstrate

this point by presenting a full-fledged prototype of checking compiler built as a customizable version of the popular gcc compiler, and called mygcc. This prototype checks and compiles full C while adding only about 1000 lines of C source code to the gcc compiler. First performance measures indicate that the overhead of checking does not exceed compilation time even on rather complex checks, and may be as small as 15% for basic checks. Thus, it now becomes possible to integrate checking at every compilation.

Based on this concrete base, we were able to estimate the potential benefits of compiler-integrated checking on the maintenance of a significant code base — part of the Linux kernel. This experiment shows that some limitations of the sporadic checking approach can be overcome using permanent checking.

The main contributions of this paper can be summarized as follows:

1. we present an approach to closely integrate checking and compiling, and we illustrate it with a full-fledged compiler (mygcc) able to check user-defined properties in addition to usual compilation
2. we discuss some difficulties of implementing the integrated approach and give solutions to these problems
3. based on the integration of our approach, we introduce the concept of permanent code checking during the whole development process, and demonstrate its usefulness on a real example
4. our approach is specially designed to be incorporated with little effort in any compiler; in particular, our checking technology is completely language-independent.

The rest of this paper is organized as follows. Section 2 presents the approach of compiler-integrated checking. Section 3 discusses implementation issues. Section 4 presents the permanent checking approach enabled by compiler-integrated checking. Section 5 discusses related work, and Section 6 concludes.

2. Compiler-integrated checking

Existing tools for user-defined program checking use various approaches, but share a common, apparently minor design choice: they are specialized tools, doing *only* program checking. There are several important consequences of this design:

- most of the tools are completely decoupled from existing development environments
- they duplicate a considerable amount of work on program parsing and analysis; this is true even for tools that achieve a superficial level of integration by being called automatically from existing IDEs or makefiles
- they afford to perform costly analyses, which make them unsuitable for daily use throughout development; at best, existing tools aim only at *scalable* analyses
- last but not least, the vast majority of programmers completely ignore their existence.

We propose to explore the challenge of integrating user-defined checks into the tool which constitutes the core of every development process: the compiler. The goal is not just to build one experimental checking compiler, but to define a methodology to integrate user checking into *any* existing compiler.

This design decision solves the above problems, and should allow a really widespread use of program checking, in order to deliver a small yet useful part of present checking technology to every programmer.

However, this design decision is indeed a challenge to put into practice, because it imposes a number of severe constraints on the implementation:

- checking should be really fast, which means not only scalable, but comparable to compilation time
- the interface should be smoothly integrated into the compiler interface, and trivial to use; ideally, the user should express many useful checks using only a few compiler options
- the implementation should not contain complex tools such as theorem provers, expression simplifiers, or complex language interpreters, so that compiler implementors could practically accept it.

In order to fulfill the above constraints, the present approach chooses a new balance between checking power and precision on one hand, versus speed and usability on the other hand. This new balance is achieved by a *minimalist interface* consisting of constrained reachability queries, able to express a small class of user properties, checkable in linear time, but covering nevertheless many useful checks.

Furthermore, to ensure widespread adoption of the methodology into any compiler, the implementation should be easily ported to any language without the need to develop complex language-specific front-ends; ideally, the whole implementation should be completely language independent. These language-independence is achieved by a *minimalist implementation* of pattern matching, using unparsed patterns.

2.1. Minimalist pattern matching

Traditionally, source code pattern matching has been reduced to tree matching, following two different approaches.

According to the first approach, patterns are expressed directly as ASTs, so that any algorithm for tree matching can be used to match them with the program AST. This approach is simple to implement, but writing patterns in AST form requires the user to be aware of both the AST representation of programs and a specific textual notation for it.

According to the second approach, patterns are expressed directly in the concrete syntax of the programing language extended to contain pattern variables (also called meta-variables to distinguish them from the variables of the underlying programming language). Writing patterns in concrete syntax is trivial for any programmer, but this approach is difficult to implement, because it requires to build a pattern parser. The pattern parser must implement an extended version of the programming language's grammar. Extending the grammar of a real programming language to allow for pattern variables is a difficult task, because it introduces shift/reduce conflicts, which are usually difficult to solve. As a result, pattern parsers usually implement a limited pattern grammar, allowing meta-variables to occur only in certain positions.

Thus, abstract syntax patterns are easy to implement but difficult to use, while concrete syntax patterns are easy to use but difficult to implement. To solve this apparent contradiction without sacrificing any of the terms, we designed a new technique of pattern matching based on unparsed patterns.

Unparsed patterns are program fragments written in the concrete syntax of a programming language where meta-variables may replace any construct that is represented as a subtree in the AST. However, unparsed patterns can be matched with ASTs without being parsed at all [33]. The key insight behind unparsed pattern matching is that when matching a program AST with a pattern represented as a string, there is enough structure information in the AST so that the pattern needs not be parsed. In fact, the pattern matching algorithm works by unparsing the AST to compare it with the pattern, rather than parsing the pattern.

By avoiding to implement a pattern parser, unparsed pattern matching is completely language-independent, except the part that unparses an AST. However unparsers for any language can be generated automatically bases on the grammar of the language. Moreover, most compiler already include an unparser for debugging purposes. As a result, unparsed pattern matching can be implemented almost for free in any compiler.

2.2. Minimalist interface

Syntax. By allowing meta-variables to stand for any subtree in the AST, unparsed patterns provide a powerful tool to express syntax information in user-defined properties. This is already sufficient to define a large class of properties related to code inspection. To go beyond that, we need to integrate control-flow information in our interface.

Control flow. It is well known that many dataflow analyses and program checks can be expressed as reachability queries over an "exploded program graph", which is the product of the program CFG and another graph (a value-flow graph, or an automaton, for instance).

Integrating control-flow in our minimalist interface is based on the observation that many sequencing properties that were successfully used in the literature to find bugs in real code can be expressed as one or several instances of simple reachability queries *directly* on the program CFG. This form of simple reachability problems, that may be called "constrained reachability queries" have the form: "Is there a path from a statement p_1 to a statement p_2 avoiding statements p_3?". In this questions, p_1, p_2, and p_3 are code patterns that define classes of program statements.

For example, looking for memory leaks can be expressed as "Is there a path from a *malloc(x)* statement to the *exit* node passing only through statements other than *free(x)*?". The exit node may be either any return statement when checking intra-procedurally, or return statements from the main function, when checking inter-procedurally.

Similarly, many other common checks may be expressed as constrained reachability queries. For instance:

- reading a closed file: from *close(f)* to *read(f,_)* avoiding *f=open(_,_)* (where "_" is the pattern matching anything)
- double lock: from *lock(x)* to *lock(x)* avoiding *unlock(x)*
- blocking operation with interrupts disabled: from *disable_interrupts()* to *blocking_function()* avoiding *enable_interrupts()*

Data flow. Using reachability in the CFG and unparsed patterns, an unexpected number of useful checks can be encoded. However, the properties thus defined lack any information on the values of program variables.

To give a concrete feeling of this limitation, let us consider a check for potential null dereferences of dynamically allocated pointers. This check can in principle be expressed as a reachability query: "Is there a path from *x=malloc()* to **x* avoiding *if(x!=0)*?". However, as it is written, the reachability query ignores the outcome of the test. However, only paths going through the "else" branch could contain potential null dereferences.

In order to take into account the result of the test, the query should avoid only *successful* tests matching the pattern *x!=0*. That is, the query has to be written more precisely as: "from *x=malloc()* to **x* avoiding *successful* tests

x!=0 and *unsuccessful* tests *x==0*". This way, dataflow information can be integrated very naturally in our minimalist interface.

Semantics. Semantic information can be easily added by complementing patterns with calls to executable functions internal to the compiler. For instance, the following pattern matches statements that allocate not enough space for the destination variable or expression:
$x = malloc(y) \mid \{const(y)\ \&\&\ val(y) < size(type(x))\}$

By calling other internal functions of the compiler, any information computed by any analysis or optimization pass (on the AST level) can be reused by the checking phase, thereby taking immediate advantage of the compiler-integrated approach.

Mixing all together. A constrained reachability query (CRQ) is a query of the form: "Is there a path from a statement p_1 to a statement p_2 avoiding: statements p_3, successful tests p_4 and unsuccessful tests p_5?". A CRQ can thus be expressed as a tuple of five patterns $\langle p_1, p_2, p_3, p_4, p_5 \rangle$. We sometimes refer to the patterns according to their role in the CRQ as: the "from" pattern, the "to" pattern, the "avoid" pattern, the "avoid-then" pattern and the "avoid-else" pattern.

Of course, some patterns can be omitted in a CRQ, to express plain reachability or even purely syntactic queries:

- integer division: [is there a path] from *int x;* to *x/_*? The "avoid" patterns missing altogether, we have a pure (or unconstrained) reachability query.
- undefined side-effect constructs: [is there a path] from *x[i++]=y[i++]* [to anywhere]? As the "to" patterns is missing, this represents a purely syntactic query looking for statements matching the pattern.

Thus, the "to" patterns defaults to the "_" pattern matching anything, while the "avoid", "avoid-then", and "avoid-else" patterns default to ϵ, the empty pattern matching nothing. The "from" pattern cannot be omitted.

This minimalist user interface has the following advantages:

- a large number of useful checks can be expressed
- checks are encoded very compactly, grouping together syntactic, semantic, control flow and data flow information
- checks are expressed very naturally from a programmer's point of view
- user properties expressed as CRQs can be checked in linear time and space, as showed below.

2.3. Checking CRQs

It is known that if meta-variables can be instantiated in negative patterns (such that the "avoid" patterns), the running time of path queries strictly increases [23]. As we strive for a class of properties checkable at compilation with an acceptable overhead, we restrict queries so that all the meta-variables must occur in the "from" pattern. This guarantees that all meta-variables are instantiated at the beginning of each path satisfying a query. Then, it is possible to check a CRQ by first instantiating the "from" patterns in a global store and then performing a traversal of the instantiated CFG.

In the first traversal, the algorithm scans all the program for "from" statements, and collects the number of different substitutions associated to them. Each such substitution represents an *instance* of the CRQ, which is then checked in two subsequent passes. The first pass over a CRQ instance collects all "from" nodes satisfying the current substitution. These constitute the initial work list for the second pass over the CRQ instance, which traverses the CFG from "from" nodes to "to" nodes visiting only nodes not satisfying the "avoid" patterns and taking only edges not eliminated by the "avoid-then" and "avoid-else" patterns. For any "to" node that is reached, there exists a path satisfying the CRQ.

The space used by the algorithm is the work list whose length is at worst the number of edges, plus one bit per CFG node to recognize already visited nodes, plus a global store for pattern variables, recording their substitution for the current instance of the CRQ. To compute the set of all instances of a CRQ during the first traversal of the algorithm, a set of global stores is needed. Therefore, the space is O(*CFG*+*CRQ*).

As each graph traversal is linear in the size of the CFG, checking one instance of a CRQ is O(*CFG*). The running time for checking all the instances of a CRQ is O(substs**CFG*), where $substs$ is the number of CRQ instances.

The precise checking algorithm is detailed in a companion paper [32], along with precise complexity analysis and a formal interpretation of CRQs as regular expressions over a labeled CFG.

3. Implementation

When implementing the compiler-integrated approach into a concrete compiler, we encountered some interesting challenges, described below.

We entirely implemented the approach described in the previous section in a prototype called mygcc built by integrating into the open-source gcc compiler the above checking (in its intra-procedural version only) and pattern matching algorithms. Mygcc is not just a checker, it is a fully

functional gcc version that performs checking as an additional compiler pass. In terms of user interface, Mygcc just adds a new gcc flag "–tree-checks=file" to specify a file containing CRQ definitions to be checked while compiling the given programs.

Due to our minimalist approach, mygcc implementation consists of only about 250 lines of modifications to existing gcc code, plus about 1000 lines of added C code, among which 600 lines implement the pattern matcher and 400 lines implement the checking engine. Mygcc is freely available both as a gcc patch and as a standalone executable [27], and we are currently working with the gcc development team to incorporate the corresponding source changes into an upcoming gcc release.

Why gcc? We chose gcc as our base because we wanted to demonstrate that the compiler-integrated approach can be easily incorporated into *any* existing compiler. This is why we eliminated from the start the idea of using a research-oriented open compiler that would have eased the task by already providing some infrastructure for extensibility and cutting-edge program analyses. As opposed to this open architecture, gcc was not meant to be extensible with user checks, and implements rather well-established analyses.

Some very important advantages of gcc are its large user base and its multiple language frond-ends for C, C++, Java, etc. By choosing gcc, we hope proving that user-defined checks can be adopted in any project, for any programming language.

3.1 Choosing the intermediate representation

The first implementation choice that we faced was that gcc has no less than four internal representations for the code. First, there are language-specific ASTs in the front-ends (C, C++, Java...). Second, these ASTs are translated by each front-end to a language-independent tree representation called GENERIC. Third, GENERIC tree are simplified to a subset of GENERIC called GIMPLE [26], that keeps some high-level information about the code (lexical scopes, control constructs such as if-then-else) but factorizes most of the syntactic variations (e.g., loops are translated into gotos). Fourth, GIMPLE is translated to a register transfer language called RTL.

Given the high-level patterns specified in user properties, the easiest choice would have been to implement checking on the first representation: language-dependent ASTs. We chose not to do so, because we aimed at a tight integration between checking and compiler analyses and transformations. Or, in order to factorize analysis code between different front-ends, most of the other high-level analysis (building of the CFG, aliases, use-defs) and high-level optimizations (tail recursion elimination, constant propagation, strength reduction, etc) are performed on GIMPLE. Therefore, to take the best advantage of being inside the compiler, we chose to implement checking on the GIMPLE representation. Another very important advantage of choosing GIMPLE is that the implemented checking can then be used on any language parsed by gcc's front-ends — provided that enough information is maintained to reconstitute the original syntax, for pattern matching purposes.

However, choosing GIMPLE significantly augments the complexity of matching high-level user patterns with desugared code.

A first slight complication is that the GIMPLE representation introduces many explicit casts not present in the source. We easily adapted pattern matching so as to ignore any casts in the AST. One consequence of this is that user patterns must not include any explicit casting.

A more serious difficulty when matching user-level patterns with GIMPLE code comes from the fact that expressions are broken down in GIMPLE to a three-address form, using temporary variables to store intermediate values. However, any use of a temporary has a previous definition in the same sequence of straight-line code. Thus, from any GIMPLE expression one can reconstitute the corresponding expression in the original program by recursively inlining the definitions of temporaries over uses. Therefore, we adapted the pattern matching mechanism such that when encountering a temporary variable use in the matched code to conceptually inline its definition before continuing the matching process.

3.2 Matching lvalues

However, inlining temporaries does not solve by itself the whole problem of matching high-level patterns with simplified code. Consider the following code fragment, which represents a downsized version of a real bug in the Linux kernel:

```
ps->t->table = malloc(sizeof(pixmap));
memcpy(ps->t->table,
       pixmap, sizeof(pixmap));
```

This fragment contains a possible null pointer dereference: the allocated pointer expression "ps->t->table" is passed unchecked to the *memcpy()* function that uses this pointer as the destination of a copy. However, when the checking algorithm is performed on the GIMPLE form to verify the CRQ "from *x =malloc(_)* to *memcpy(x,_,_)* avoid *x=_* avoid-then *x!=0* avoid-else *x==0*", the code fragment has been rewritten as follows:

```
1.  D.2208 = ps->t;
2.  D.2209 = malloc (40);
```

```
3.  D.2210 = (char *) D.2209;
4.  D.2208->table = D.2210;
5.  D.2208 = ps->t;
6.  D.2211 = D.2208->table;
7.  memcpy (D.2211, &pixmap, 40);
```

In the above GIMPLE form, the destination of the *malloc()* call is a temporary variable (*D.2209*) that does not occur at all in the *memcpy()* call, even when inlining the definition of temporary *D.2211*.

In fact, the pointer expression "ps->t->table" that was shared in the original program between the calls to *malloc()* and *memcpy()* can be found as the LHS of line 4. It is precisely this expression that should be caught by the pattern variable x, rather than the temporary *D.2209*.

The key observation here is that a GIMPLE temporary definition never represents a definition in the original program, but always represents a use in the original program. Definitions in the original program are always translated in GIMPLE as assignments to a *non-temporary*. Thus, by applying inlining of temporary uses to every statement that is not a temporary definition, one can reconstitute all the statements in the original program. In our example, the original assignment involving *malloc()* can be reconstituted from the definition on line 4, the only non-temporary definition.

Therefore, we adapted the matching algorithm to skip temporary definitions when looking for user-defined statement patterns.

3.3 Binding context

There is another interaction between temporary variables and patterns, which comes from the fact that temporary variables only have meaning when associated with the statement where they occur. When a temporary is bound to a pattern variable x, the binding should include not only the temporary, but also the context statement where it was bound. Thus, when the same pattern variable x is subsequently used in another pattern for matching another statement, inlining the temporary bound to x must be done with respect to its binding context, and not with respect to the current statement. To implement this mechanism, the global store includes for every pattern variable both its value and its binding context.

4 Permanent checking

The compiler-integrated approach, as implemented in mygcc, makes it possible to perform permanent checking during all the development process. This technique was not possible using previous checking tools.

In order to assess and validate our approach, we applied mygcc to reproduce the detection of some previously reported bugs in the Linux kernel [9]. That previous study applied 12 different user-defined checkers written in Metal for the MC tool to detect over 500 bugs in kernel version 2.4.1. All these bugs were manually inspected and/or confirmed by kernel developers. A summary of the MC results is freely accessible as an on-line database [25]. As MC is a basically intra-procedural tool, this study proved that it is indeed worth to carefully check intra-procedural properties in real system software. This also meant that even our current implementation of mygcc could have a real potential usefulness.

Using this excellent and well-established testbed, our approach had to be validated in several respects: expressiveness, precision, usefulness, scalability, and performance.

To assess the expressiveness of our properties, we expressed as CRQs as much as possible of the 12 MC checkers cited above. Out of these 12 checkers, 11 can be expressed partially or completely as CRQs: 9 using only syntactic patterns and 2 checkers using semantic patterns. Thus, a single checker cannot be expressed conveniently as a CRQ because it encodes complex checks about pointer uses and makes deductions based on these uses.

To test the precision of mygcc, we chose 95 source files in the kernel that contained 134 bugs detected by three different Metal checkers: the "NULL" checker, looking for possible null pointer de-references, the "FREE" checker, looking for uses of freed pointers, and the "LOCK" checker, detecting calls to blocking functions with interrupts disabled or while holding a spin lock. We rewrote these Metal checkers as CRQs, and we tried to reproduce as much as possible of the bugs that were reported by them. Mygcc successfully found 130 NULL bugs out of the 134 bugs found by MC. In spite of its minimal interface and implementation, mygcc missed only four NULL bugs and generated only two additional false positives. Among the bugs found by both MC and mygcc, two were diagnosed slightly differently. On the other hand, mygcc found four new bugs, not previously reported.

4.1 Need for permanent checking

In order to estimate the usefulness of permanent checking, we verified whether all the bugs reported by the MC study have been fixed in a recent kernel version, v. 2.6.13, released in August 2005. The results on kernel 2.4.1 were published in 2001 and Linux kernel developers were informed about the existing on-line database summarizing the bugs. Therefore, this experiment covers a lapse of 4 years of active maintenance of a significant code base.

When re-conducting the checks described in the previous subsection on the new kernel version, mygcc found four surviving bugs:

- one of the 4 new bugs mentioned above (in file anode.c, function *hpfs_add_sector_to_btree*) has survived

File	Checkers	Time (secs)	Overhead (%)
inode.c	none	0.291	
	NULL	0.503	72
	all	0.506	74
comx-proto-fr.c	none	0.626	
	FREE	0.715	14
	NULL	0.929	48
	all	0.989	58
iphase.c	none	2.026	
	FREE	2.242	11
	LOCK	2.309	14
	NULL	3.644	80
	all	4.013	98

Table 1. Performance of mygcc.

identically; the containing function has only slightly changed in the mean time; interestingly, the other 3 previously unreported bugs have disappeared

- one bug (in file skfddi.c, function *skfp_driver_init*) has survived identically; the function has only slightly changed in the mean time
- one bug (in file riotable.c, function RIOReMapPorts) remained untouched, in spite of the fact that this function has been significantly changed for other reasons
- one bug (in file inode.c function *bfs_read_super*) survived in a different form; the code has radically changed between the two versions: the containing function does not exist anymore, but a code fragment similar to the old bug can be found now in another function (*bfs_fill_super*)

The fact that almost all the bugs were fixed in the new version clearly shows that the MC report was taken very seriously into account by Linux kernel developers. The three previously reported bugs that survived in spite of this intense correction effort demonstrate that without a proper tool to enforce user properties, even well-known bugs can survive for long periods of time (four years in our case), or can be re-introduced during maintenance.

4.2 Performance

The examples described in the Linux study illustrate the fact that mygcc is able to check any program that gcc can compile. Thus, the scalability of the prototype to real programs is clearly demonstrated. But mygcc aims not just at being scalable to large programs, but to impose a reasonable overhead on compilation time.

We measured the overhead of different checkers when compiling three programs that are part of the above Linux study: a program featuring only NULL bugs, one with additional FREE bugs, and a last one with the three types of bugs we checked for. The results are summarized in Table 1.

The benchmarks were performed on a Linux PC with an Athlon XP2800+ processor and 256MB of memory.

The checking overhead is directly related to the number of checkers used, to the number of CRQ instances found in the program, and to the size of the patterns. This explains the large variations between the different measuring points. However, it can be seen that checking time never exceeds compilation time in these typical examples of the Linux study. Overheads are of the order of 10-15% for a very simple checker (FREE, containing a total of 6 patterns), 15% for a moderate checker (LOCK, including 11 patterns), and 50-80% for a complex checker (NULL, including a total of 51 patterns, among which 24 are disjuncts of a single "from" pattern). The maximum overhead when combining all the checkers is 98%.

When interpreting the figures, it is important to note that we did not have the time to optimize the implementation of the current prototype. To give only one example, mygcc internally uses pattern matching to decide whether a node is an assignment; while this self-application is elegant, this check could be optimized to check the AST node label.

5 Related work

The most common approach to user-defined checking is to define a programming model in which users may write their own program inspection passes. Tools implementing this approach incorporate a front-end that parses the program in the form of an AST and offer either a application programming interface (API) or a domain-specific programming language (DSL) to walk the AST and implement different forms of checks.

API-based code inspectors include SoftBench CodeAdvisor from HP, or Checkstyle [7], in which user-defined checks have to be coded in C++, respectively in Java. More recently, some extensible code inspectors such as PMD [29] build an XML representation of the AST, on which user-defined checks can be expressed either in JAVA, or in a declarative way using Xpath patterns. API-based tools allow in theory to implement any user-defined checks. They offer a solid basis to inspect syntax, but little or no semantic information is pre-computed. None of these tools precompute the control-flow graph, therefore no dataflow information is available. For these reasons, API-based code inspectors make it easy to define syntax checks such as adherence to a coding standard, or computations based on syntax traversal such as function call graphs or class hierarchy extraction. In turn, writing any kind of non-local semantic checks such as verifying sequences of operations or performing model checking requires a significant amount of

code.

Tools defining a DSL to write code checkers include CodeCheck [1], tawk [20], defining an imperative language close to C, Genoa [13] defining a functional language close to Lisp, and ASTLOG [11], defining a variant of the Prolog language. DSL-based tools can very compactly encode sophisticated tree patterns or tree traversals, but none of the cited DSLs integrate control or dataflow information in the language, neither in explicit nor implicit form.

Writing checkers for both API and DSL code inspectors requires the user to be aware of the details of the AST representation for the subject language, in addition to the API or DSL to traverse it.

Another set of program checkers such as Splint [16, 30] and CQUAL [17] are based on extensible type checking. In this approach users must annotate the types of the programing with qualifiers in order to express new classes of program properties that can be checked automatically. Type-based checkers are very efficient (e.g., linear-time) and precise (e.g., sound) in verifying "global" properties in a program, i.e., that do not depend on control flow. Some of these checks could definitely be integrated in a compiler-integrated approach, but for now are implemented as standalone tools. Some extensions were added to check flow-sensitive types [18], but in this case the performance is no more suitable for permanent checking.

Yet another class of extensible checkers transpose model checking techniques, used since a long time in hardware verification, to programs, viewed simply as CFGs, in which the semantics of individual program statements is usually ignored. In this approach of lightweight model checking, user-defined properties represent legal sequences of operations, and are represented by finite automata. Transitions are triggered by syntactic patterns matching program statements. Checking is done by conceptually executing the automata along the CFG. Lightweight model checkers include Cesar [28] for checking Fortran and his evolution called FLAVERS [10] for Ada and Java, MC [14] for C and its variant MJ [3] for Java, MOPS [8] for C, PQL [24] for Java and CodeSurfer Path Inspector [19] for C. Engler et al. clearly demonstrated the practical usefulness of the lightweight model checking approach, by applying MC to detect hundreds of system programming bugs [15] and security bugs [2] in C code. The running time complexity of these tools has been precisely analyzed in the framework of parametric regular path queries [23]. Essentially, the checking time depends on the size of the user automaton. For simple automata, checking may be done in linear time. In fact, the checks allowed by our tool are a particular case of lightweight model checks where the automaton has a fixed form with only three states: the initial state, the state after a "from" node, and the error state. One original feature of mygcc it allowing to define transitions that depend on variable values, using the "avoid-then" and "avoid-else" patterns. More importantly, all cited tools are distinct from the compiler, and therefore duplicate a great amount of analysis work. Note also that our unrestricted, language-independent pattern matching could be useful in many of these tools.

More precise program checkers take into account variable values in order to distinguish between feasible and unfeasible path. Among them, SLAM [4], Blast [21], and ESP [12] also express user properties as automata. The BLAST checking algorithm has been integrated within an existing IDE as an Eclipse plug-in and optimized to work incrementally, in order to support "Extreme model checking" [22], which consists of performing user checks on each release of a program, during software development. These tools integrate complex subsystems such as symbolic executors, theorem provers and/or expression simplifiers, that cannot reasonably be integrated within compilers. This means that they are designed to remain standalone tools, used out of the critical path in development. Our approach for permanent checking is complementary to extreme model checking, as it chooses to perform simpler checks but that can be integrated easily in every compiler and re-done at every compilation.

Finally, full software model checkers, such as, for example Java PathFinder [31], verify user-defined properties (possibly defined as automata) on non-deterministic programs by trying to execute all possible sequences of non-deterministic choices. The applicability of this approach is usually limited to medium-sized systems because of the state space explosion problem. This remains true even if existing tools achieve great savings by using complex search space strategies, by reducing the number of states explored, and by reducing the state storage cost.

A quite different approach to program checking is to express used-defined properties as so-called contracts associated to interfaces, and consisting in predicates to be checked before and after function calls. Thus, JML [6] extends Java with contracts expressed as stylized comments, and Spec# [5] extends C# with contracts integrated in the base language. In both of these systems, user properties cover a restricted set of first-order logic, and may be checked either statically or dynamically. Similar to our approach, user properties are handled by an extended compiler. Moreover, some of the properties mentioned in this paper can be re-phrased as function contracts. One major distinction is that contracts are tied in these systems to pre- and post-conditions around function (or method) calls, while mygcc can check properties on every program construct, using pattern matching — not only on function calls. Besides, contract properties cannot directly refer to the control flow, which is a very natural way to express some properties. On the other hand, Spec# defines non-null types that are stati-

cally checked, which allows to protect against null pointer de-references. These types are similar to CQUAL's type qualifiers, discussed above.

6 Conclusion

We presented a pragmatic approach for easily extending existing compilers with user-defined checks, very simple to express and very efficiently checked. The practical applicability of the approach is demonstrated by its very concise implementation in the gcc compiler.

The fusion between checking and compiling enables a software development method in which checking permanently accompanies evolution, from the early coding phase to the maintenance phase. It also considerably increases efficiency by eliminating a lot of duplicated analyses.

Beyond these immediate advantages, the fusion between the checker and the compiler opens up new perspectives for:

- Integrating specifications within library interfaces, with no language extension: each interface file could be complemented by a separate check file describing sequencing constraints.
- Integrating specifications within the program itself: since the compiler is also the checker, one can imagine to define, enable, and disable user checks using compiler pragmas.
- Integrating checking with program analyses and optimizations: by implementing all in the same tool, cross-fertilizations are possible, such as optimizations dependent on sequencing constraints, or checks enabled by program optimizations.
- Integrating checking within generated code: checks that cannot be performed statically by the compiler could be easily integrated into the generated code to be executed at runtime.

Mygcc is a particular point in a wide spectrum of possible trade-offs. It proves that it is possible to integrate compilation and checking, almost for free in terms of implementation, and with an acceptable overhead. But this is just a starting point. One open question that is raised is: how much checking power can one put in a compiler to maintain a reasonable runtime and implementation overhead?

References

[1] Abraxas Software, Inc. CodeCheck. http://www.abxsoft.com

[2] K. Ashcraft, D. Engler. "Using Programmer-Written Compiler Extensions to Catch Security Holes". In Proc. IEEE Symp. on Security and Privacy. May 2002.

[3] G. Back, D. Engler. "MJ - a system for constructing bug-finding analyses for Java". Technical report, Stanford University. September 2003.

[4] T. Ball, S. Rajamani. "The SLAM Toolkit". In Proceedings of the 13th International Conference on Computer Aided Verification. LNCS Vol. 2102. 2001.

[5] M. Barnett, K. Leino, and Wolfram Schulte. In CASSIS 2004, LNCS vol. 3362, Springer, 2004.

[6] L. Burdy,Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, E. Poll. "An overview of JML tools and applications." In T. Arts, W. Fokkink, eds.: "Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)". Volume 80 of Electronic Notes in Theoretical Computer Science (ENTCS)., Elsevier, 2003.

[7] Checkstyle. Open-source project at SourceForge.net. http://checkstyle.sourceforge.net

[8] H. Chen, D. Wagner. "MOPS: an infrastructure for examining security properties of software". In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS). Washington, DC. November 2002.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler, "An empirical study of operating system errors". In 18th Symp. Operating Systems Principles (SOSP). Oct 2001.

[10] J. Cobleigh, L. Clarke, L. Osterweil. "FLAVERS: A finite state verification technique for software systems". IBM Systems Journal, 41(1). 2002.

[11] R. Crew. "ASTLOG: A Language for Examining Abstract Syntax Trees". In USENIX Conference on Domain-Specific Languages. October 1997.

[12] M. Das, S. Lerner, M. Seigle. "Esp: Path-sensitive program verification in polynomial time". In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), Jan. 2002.

[13] P. Devanbu. "GENOA — a customizable, front-end-retargetable source code analysis framework". ACM Transactions on Software Engineering and Methodology (TOSEM) 8(2). April 1999.

[14] D. Engler, B. Chelf, A. Chou, S. Hallem. "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions". Proc. of 4th Symposium on Operating System Design and Implementation (OSDI), San Diego. October 2000.

[15] D. Engler, B. Chelf, A. Chou, S. Hallem. “Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions”. Proc. of 4th Symposium on Operating System Design and Implementation (OSDI), San Diego. October 2000.

[16] D. Evans, D. Larochelle. “Improving Security Using Extensible Lightweight Static Analysis”. IEEE Software 19(1). January 2002.

[17] J. Foster, M. Fhndrich, A. Aiken. “A Theory of Type Qualifiers”. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Atlanta, Georgia. May 1999.

[18] J. Foster, T. Terauchi, A. Aiken. “Flow-Sensitive Type Qualifiers”. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Berlin, Germany. June 2002.

[19] Gramma Tech. CodeSurfer Path Inspector. http://www.grammatech.com

[20] W. Griswold, D. Atkinson, C. McCurdy. “Fast, Flexible Syntactic Pattern Matching and Processing”. In 4th International Workshop on Program Comprehension. 1996.

[21] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, W. Weimer. “Temporal-Safety Proofs for Systems Code”. Proc. of the 14th International Conference on Computer-Aided Verification (CAV). LNCS 2404. Springer-Verlag, 2002.

[22] T. Henzinger, R. Jhala, R. Majumdar, M. Sanvido. “Extreme model checking”. In Proceedings of the International Symposium on Verification: Theory and Practice. LNCS 2772. Springer-Verlag, 2004.

[23] Y. Liu, T. Rothamel, F. Yu, S. Stoller, N. Hu. “Parametric regular path queries”. ACM SIGPLAN Notices, 39(6) (PLDI). May 2004.

[24] M. Martin, B. Livshits, M. Lam. “Finding application errors and security flaws using PQL: a program query language”. In Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA). 2005.

[25] MC bug viewer. http://metacomp.stanford.edu

[26] J. Merill, “GENERIC and GIMPLE: A New Tree Representation for Entire Functions”. Proc. of the GCC 2003 Summit.

[27] Mygcc prototype. http://mygcc.free.fr

[28] K. Olender , L. Osterweil. “Cesar: a static sequencing constraint analyzer”. ACM SIGSOFT Software Engineering Notes 14(8). December 1989.

[29] PMD. Open-source project at SourceForge.net. http://pmd.sourceforge.net/

[30] Splint. Open-source project. http://www.splint.org

[31] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. “Model Checking Programs”. Automated Software Engineering Journal, 10(2), April 2003.

[32] N. Volanschi. “Condate: A Proto-language at the Confluence Between Checking and Compiling”. Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP). 2006. To appear.

[33] N. Volanschi. “Unparsed patterns: integrating unrestricted, concrete syntax pattern matching in any compiler”. Technical Report, January 2006. Available from [27]