Metal: a language and system for building lightweight, system-specific software checkers, analyzers, and optimizers

Andy Chou and Dawson Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
{acc,engler}@cs.stanford.edu

1 Introduction

Frequently, systems require that programmers obey intricate rules for both correctness and performance. Common examples include admonishments that: "accesses to variable A must be guarded by lock B," "system calls must check users pointers for validity before using them," "to avoid stack overflow, kernel code must not allocate large local variables," "interrupts must be disabled before accessing state shared with a device driver," "message handlers should free their buffers as quickly as possible to allow greater parallelism." While a single violation of these rules can degrade performance or even crash the system, they are typically unchecked, existing only on paper or in the implementor's mind. As a result, violations must be manually caught, and optimizations based on semantics manually performed. The first situation leads to uncaught errors, the second to lost opportunities.

Many of these restrictions could be checked and exploited by a compiler. For example, many of them define temporal relations — files must be opened before being read from, interrupts must be enabled after being disabled — that are easily checked by observing the order of function calls. The main barrier for a compiler checking or optimizing at this level is that while it must have a clear understanding of the semantics of its input code, it typically has no idea of the "meta" semantics of the system that this code is used to construct. As a result, properties inexpressible (or just not expressed) in terms of the underlying language's type system are not checked. And, while the compiler will optimize code at the low-level of programming constructs, its lack of knowledge of higher-level meaning places a hard barrier on its effectiveness. For example, consider an application that makes heavy use of message passing: the compiler may generate hyper-efficient code for argument marshaling and checking return results but, if it does not understand what a message send is, it cannot realize that two abutting sends could be coalesced, or that a send for a send/reply pair may be pushed earlier in execution, etc.

As a result, existing practice has an unfortunate dichotomy. Implementors understand the semantics of the systems they build and use but do not have the mechanisms to check or exploit these semantics automatically. Compilers do have the machinery to do so, but their domain ignorance prevents them from using it. This paper presents a language, *metal*, that attacks this problem by making it easy for system implementors to write lightweight domain- or even application-specific checkers and optimizers. These are expressed as high-level compiler extensions that are written as general State Machines (SMs) and are applied to the programs consumed by the compiler. Extensions use patterns expressed in an extended form of the base language to search for interesting features and transition between states on pattern matches. Extensions can use arbitrary code to compute their results and transitions — *metal*'s state machine framework is used intended to make extensions easy to write, it does not restrict what they can do. In our system, *metal* programs are compiled and dynamically linked into our extensible compiler, *xg++*, at compile time. They are invoked after the compiler parses and builds up the Abstract Syntax Tree (AST) for each input function and applied to every path through the function.

As an example, an extension to check that kernel routines that disable interrupts always re-enable them could define states for "interrupts enabled" and "interrupts disabled" and then use patterns describing the function calls that perform these acts to transition between states, giving errors for duplicate enable/disable and paths out of the procedure that leave the state machine in an "interrupts disabled" state. The code for this extension is given in Section 2.

Metal is not a "boutique" language. We use it on a daily basis and have built checkers that have caught system killing errors in domains ranging from the FLASH message passing machine's embedded cache coherence software [8], synchronization errors in Linux device drivers [4], and user pointer errors in the Xok exokernel [4]. System-specific *metal* optimizers have been used to specialize memory allocator calls to the size of the allocation, replace copying I/O operations with non-copying memory mapped versions, and optimize the system itself [5]. In our experience, many interesting system properties can be captured in simple sentences that can be checked with small *metal* extensions (i.e., < 100 lines of code).

This paper makes three main contributions:

1. The *metal* language. *Metal*'s approach to specifying code is novel, allowing it to be high-level and powerful yet easy to implement and largely language independent.

2. An implementation of *metal* based on the the ex-

tensible *xg++* compiler, which in turn is based on the GNU g++ compiler. *Metal* is designed to be easy to implement in existing compilers: its implementation in the context of one of the world's most complex is a reasonable demonstration that it is.

3. A variety of examples that show the power of the approach; many of these have been used to optimize and detect errors in real systems.

The paper's structure is as follows. Section 2 gives an overview of the language. Section 3 explores the language's main interesting linguistic feature, its mechanism for expressing patterns. Section 4 discusses the use of path variables to maintain per-path data while performing an analysis. Section 5 discusses implementation issues with patterns. Section 6 discusses implementation issues with path variables. Section 7 contains some applications and experience with *metal*, Section 8 describes related work, and Section 9 concludes.

2 Language Overview

Metal programs are structured as state machines that express program analysis passes using states, transitions between states, patterns that trigger transitions, and actions to be performed when a transition is triggered. The state machine part of the language is straightforward and can be viewed as syntactically similar to a "yacc" specification. Users can create arbitrary states, transition between them, and perform actions that are escapes into the base language.

The main novelties of the language are patterns and the *metal* program execution model. Patterns are written in an extended version of the base language (C++ in our implementation), and can specify (almost) arbitrary language constructs such as declarations, expressions, and statements. They are used to specify when an SM makes a transition. Using the base language as a way to express patterns allows the patterns to be powerful yet easy to use, since they closely mirror the source constructs they are searching for, and easy to implement, since they can be parsed using the underlying base compiler.

The extension below illustrates many of *metal*'s features. It checks that procedures in the Linux kernel that disable interrupts (using a call to `cli()`) subsequently re-enable them on all paths (using a call to `sti()`). It gives an error on duplicate calls to these functions, or if the procedure appears to exit with interrupts disabled. (A slightly more complex version of this checker found errors in several Linux device drivers [4].)

```
{ #include "linux-includes.h" }
sm check_interrupts {
 /* initial state: function has interrupts enabled */
 enabled:  { cli(); } ==> disabled
         | { sti(); } ==>
                { err("intr's already enabled"); }
         ;
 disabled: { sti(); } ==> enabled
         | { cli(); } ==>
            { err("intr's already disabled"); }
         | $end_of_path$ ==>
   { err("exiting with intr's disabled!"); }
    ;
   /* stop traversing this a path if we hit a panic or
      abort (either kills the kernel) */
   decl { char * } any_str;
   all: { panic(any_str); } | { abort(); } ==>
                { mgk_return(MGK_STOP); } ;
}
```

The *metal* system applies client SMs down every possible code path in a function. (The system attempts to, but does not guarantee that it will omit impossible paths). The given SM uses three states, `enabled`, `disabled`, and `all` to track the current interrupt state for each path. By default, SMs start in the state mentioned in the first transition definition (here `enable`). As it begins in this state, the system checks in lexical order to see if any of its patterns match in order, if so, "consumes" the input that matched, executes the optionally provided action and sets the state to the optionally specified new state (the token after the `==>` operator).

For the above SM, `enable` searches for all interrupt disabling using the literal pattern "`{ cli(); }`," which, if it succeeds, causes the SM instance on this path to transition to the `disabled` state. Similarly, calls to `sti()` are caught and cause an error (multiple enablings are questionable at best). By omitting the state token for this transition, the SM will remain in the current state. (Additionally, the transition can be manually overridden within the action using an explicit state assignment.) Both patterns and actions are surrounded by braces. Patterns are written in extended version of C, while actions can be arbitrary C code. Since patterns and actions opaque to the language, they could easily be other languages as well.

The `disable` state is roughly the inverse of `enable`, with the refinement that it uses the special pattern `$end_of_path$` to detect then the end of current path is reached in this state. Other special patterns and tokens are given in Figure 1.

The final state `all`, is a predefined by *metal* and used to specify transitions that can happen in any state. The alternative would be writing the transition in every rule. These default transitions are only run after all lexically preceding rules have failed to match. The SM uses this state along with a transition to the special `$stop$` state (which stops the traversal of a given path) to eliminate false positives by stopping a path traversal if it hits a function that will cause the kernel to self-destruct, since such paths are likely to not obey system restrictions.

All SMs are considered to accepting by default; rejecting states should define a transition with an empty pattern and an action that calls an error function.

As shown, the language has two pieces: (1) pattern matching and (2) the SM machinery. The next two sections discuss each separately in more detail.

3 Patterns

Patterns allow an extension to match code fragments in a simple and reliable way. The most primitive pattern is one that exactly matches the code that the extension

Special types	Meaning
any_expr	any legal expression
any_scalar	any legal scalar
any_ptr	any legal pointer
any_stmt	any legal statement
any_args	any set of function arguments
any_call	any function call
Special states	**Meaning**
$stop$	stop traversing a path
all	always active state
Special patterns	**Meaning**
end_of_path	match the end of a path
$epsilon$	matches any token
end_of_sm	matches when the SM has been applied over the entire function

Table 1: Representative *metal* keywords and their meaning.

wishes to recognize. Spaces and other lexical artifacts don't interfere with matching:

```
{ cli(); } // matches  'cli();'  and  'cli ();'
{ rand() } // matches in 'x = rand()', 'foo(rand())'
```

SM "hole variables" allow patterns to contain wildcards where any subtree of the appropriate type will match. Hole variables are declared in the body of an SM using the type qualifier `decl`:

```
decl { void * } ptr;
decl { float } f;
decl { const int } sz;
```

These declarations make the variable `ptr` a hole that matches any expression of type `void *`; likewise `f` matches any expression of type `float`. The type `const int` for `sz` matches compile-time constants of type `int`, which includes constant literals and anything that can be folded into a constant at compile time. In particular, this matches uses of `sizeof` in C and C++.

In addition to supporting pattern parameterization, hole variables allow matching to be type driven. For example, the following *metal* extension can be used to verify that embedded or OS code does not contain any references to floating point variables (they typically cannot because of convention, or because there is no FPU):

```
sm no_float flow_insensitive {
    decl { float } f;
    decl { double } d;
    all: { f } | { d } ==>
       { err("No floating point in kernel code"); }
       ;
}
```

The patterns are simply holes, which will any expression of type `float` or `double`. It also matches variable declarations of those types. In general, transitions may occur on any one of a set of patterns that are separated with the "|" operator. Each pattern will be applied in order and the action run when matched. As a slight refinement, the SM is specified to be "flow-insensitive" which means that it is applied over the function ignoring control flow. This improves the speed of the extension and also guarantees that errors will only be reported a single time.

A strength of patterns is that they closely resemble what they match. A weakness of this literalness is that they may be either over specific (by introducing extraneous detail) or under specific (by not allowing restrictions to be expressed). *Metal* mainly gives programmers control over what is matched by introducing new types that can be used to broaden holes to an entire class of related types (e.g., all pointers or function calls), or restrict them to a more narrow subset (e.g., all integer variable declarations rather than all integer types).

```
decl { any } p;    // matches any expression
decl { scalar } s; // matches any scalar expression
```

As we show in Section 4 these types can be used to check type-independent properties such as checking that variables are initialized before use.

A hole variable used within an action (as opposed to a pattern) refers to the *tree* matching the hole. We chose these semantics for hole variables to allow the extension to examine the matched part of the tree using a set of library functions provided by *xg++*. The following code shows an SM that uses hole variables and several `xg++` library functions to check for under allocation errors with the C `malloc` memory allocator by checking that the `sz` is greater than or equal to the size of the object pointed to by `ptr`, according to its type.

```
sm check_malloc, flow_insensitive {
  decl { const scalar } sz;
  decl { any } p;

  start: { p = (any)malloc(sz) } ==>
    {
      int got = mgk_int_const(sz);
      int need = mgk_sizeof(mgk_deref(p));
      char *name = mgk_var_name(p);

      if(got < need)
         err("allocating %d bytes, %s needs %d\n",
                got, name, needs);
    }
}
```

Like the `no_float` SM, this SM checks a static property of certain calls to `malloc` that doesn't depend on the path taken to reach the call. The runtime system function `mgk_int_const` is used to get the value of the constant stored in `sz`'s corresponding AST tree. Similarly, the action uses `mgk_deref` to dereference the allocated pointer to get its base type and then applies `mgk_sizeof` to get the size of this type.

This checker shows a practical stylized way to use *metal*'s extended types to make third-party interface checkers portable — while many implementations of standardized functions get the function names right, their parameter types are less regularly correct. For example, `malloc`'s size argument may be a signed integer on one installation, an unsigned on another. Exact typing would prevent the checker from recognizing uses portably.

If a hole variable appears multiple times in the same pattern, both instances must contain the same code. The following SM finds all identical abutting statements:

```
sm sm_statements {
  decl { any_stmt } s;

  start: { stmt(s); stmt(s); }
         ==> { err("duplicate statements"); }
       ;
}
```

Statements are of type any_stmt, and the special form stmt(s) will match any statement or compound statement and place the result in s. On the following code the extension will match the two assignments at lines 1 and 2, as well as the first two calls to foo() at lines 4 and 5. Since successful matches consume their associated trees, the pairing of calls at line 5 and 6 will not match.

```
1: a = 1;  /* match 2 */
2: a = 1;
3:
4: foo();
5: foo(); /* match 1 */
6: foo();
```

3.1 Pattern variables

In the malloc example, the pattern was specified directly as part of a transition. If several transitions use the same pattern, it may be more convenient to use pattern variables:

```
pat malloc = { p = (any)malloc(sz) };

start: malloc ==> { ... }
             ;
```

The holes can still be used in the action of the transition just as if the pattern were specified in the transition itself. The "|" operator works with pattern variables in transitions as well as when defining pattern variables in terms of other variables:

```
pat p1 = { ... };
pat p2 = { ... };
pat p3 = p1 | p2;

// the following are equivalent
state: p1 | p2 ==> X ;
state: p3 ==> X ;
```

Pattern variables cannot be parameterized. They are intended to be used as lightweight sugar rather than syntactically-general (unusual) functions.

4 State variations

4.1 Path variables

Often it is useful to have variables which are unique to each path. We introduce a qualifier, path, to mark path variables:

```
path { int } x;
```

This causes the value of x to be saved and restored as the SM is pushed up and down paths. Path variables make it easier for the SM to maintain the illusion that it is only checking a linear code path. Information in path variables only flows down: before an SM traverses down to the next level on a path, the values of all path variables at this level are saved. When control returns back to this level, these values are restored. Thus, while modifications done to a path variable can be seen by lower nodes on a given path, any mutations by these nodes disappear. Without this ability, SM actions would have to remember to save and restore state as their SMs traversed paths.

As a simple example, we make the linux interrupt checker from Section 2 aware of interrupt flags. Interrupt flags are local variables used to save and restore the state of the interrupts; a common idiom in Linux is to save the interrupt state with a flag variable, disable interrupts for a critical section, and then restore the interrupts to the state saved in the flag variable. The checker only works for a single flag variable and it misses certain errors, such as restoring flags without first saving them. Despite these limitations, this checker is practical and is capable of finding interrupt state bugs in Linux; an extended version of this checker found 5 potential bugs in device drivers and networking code.

```
{ #include "linux-includes.h" }
sm check_interrupts {
  /* initial state: function has interrupts enabled */
  enabled:  ... // same as previous example
  disabled: ... // same as previous example

  decl { char * } any_string;
  all: ... // same as previous example

  /* save/restore interrupt state */
  decl { unsigned } flags;
  path { state_t } saved_intr;
  all: { save_flags(flags); } ==>
              { saved_intr = mgk_state; }
     | { restore_flags(flags); } ==>
              { mgk_state = saved_intr; }
     ;
}
```

The mysterious variable mgk_state and type state_t represent the state of the SM (i.e. either enabled or disabled) and the type of mgk_state, respectively.

Metal also has "state variables," which are almost identical to path variables with type state_t and can be declared using the state qualifier:

```
state y;
```

A subtle point is that mgk_state is itself a state variable implemented like y internally, except that it is implicitly added to every SM.

State variables differ from path variables with type mgk_state only in that they can be used in transitions with the "var.state" notation:

```
// mgk_state is an implicit state variable;
// the following two are equivalent
state1: ... ==> state2;
mgk_state.state1: ... ==> mgk_state.state2;

// other state variables must be qualified
state y;
y.state1: ... ==> y.state2;
```

Each state variable undergoes transitions independently. If a pattern matches for a transition rule of one state variable, it does not preclude another pattern match for a transition rule of another state variable.

4.2 Coupling states to variables

In all of our examples, there has been global SM state that is coupled to each path: it is modified as we go down each path, and then restored to its previous value upon backtracking. In many situations, it is more appropriate to have states coupled to variables (or arbitrary nodes) rather than control paths.

Metal provides per-variable state by extending the state variable notation "`var.state`" to hole variables declared with the qualifier "`state decl`." A `var.state` expression can be used as a transition (to indicate that the given variable should be placed in the specified state) or as a rule state (to indicate that if the hole variable is in these states that it should be run on the given patterns). For example, the following fragment binds the hole representing a lock variable `l` to the state `locked` for each instance of the pattern { `lock(l)`}:

```
  state decl { lock_t } l;

  start: { lock(l) } ==> l.locked
     | { unlock(l) } ==>
           { err("not locked"); }
     ;
   ...
}
```

We can then associate transitions with each instance of `l.locked` as follows:

```
  l.locked:
       { lock(l) } ==>
           err("already locked");
     | { unlock(l) } ==>
           l.$stop$
     | $end_of_path$ ==>
         { err("lock not released"); }
     ;
```

There are two things to note here. First, if the hole variable `l` appears in pattern bound to a state guarded by it, then the same tree that matched the initial instance must be matched there. Thus, the hole variable `l` in `unlock(l)` for the state `l.locked` will only match trees identical to that which created this `l.locked` instance in the `start` state. For example, if we apply the above SM to the following code:

```
int foo(lock_t l1, lock_t l2) {
        lock(l1);

        unlock(l2);
}
```

The `lock(l1)` call will trigger the creation of a new state bound to the tree representing `l1`. The subsequent `unlock(l2)` will not cause this binding to transition, since the tree will not match. Our naive SM will thus give two errors when run on this code: first that `l1` is not released, second that `l2` was not locked.

4.3 Computing Dominators: All path vs any path analysis

For the most part, our examples have been meta-level checkers rather than optimizers. Since checkers typically look for any path that has a violation, a general depth first style search down all paths works well. Optimizers on the other hand typically need to make sure that a condition holds on all paths in order to replace or transform an operation. A stylized way to compute such domination information in *metal* is to use two SM passes. The first pass marks tree nodes with the condition of interest, and the second pass checks if all tree annotations are the same.

As an example, the following SM is a drastically reduced version of one used to check embedded software in the FLASH shared memory machine for inefficient message header manipulations. This software is constructed as a set of message handlers, each handler receives incoming cache coherence messages for which it may send several responses. To do so it must a set the current message buffer opcode to the correct type. It can save an instruction by xor'ing the current value in the message buffer with the value it wants to:

```
// costs two cycles
dh.opcode = OP_NAK;
// costs 1 cycle (works if OP_RPLY in message)
dh.opcode ^= OP_RPLY ^ OP_NAK;
```

Since FLASH code is implementing functionality (cache coherence) normally done by hardware, saving every instruction possible is important. (The code has hundreds of these operations done by hand.) Our simple checker is advisory, providing hints when its analysis indicates that an optimization should be possible. The checker starts with an unknown opcode value and enters a known state after the first opcode assignment. If an opcode assignment is reached with the same current opcode value along every path, the checker alerts the user that xor could be used to save an instruction.

```
state decl { const int } op;
pat asgn = { dh.opcode = op };

/* pass 1 */
sm xor_mark {
  unknown: asgn ==> known
     ;
  known: asgn ==>
       { mgk_dom_annot(op, mgk_int_const(op)); }
     ;
}

/* pass 2 */
sm xor_sweep, flow_insensitive {
  start: asgn ==>
     {
       int orig;
       /* fails if wasn't uniquely annotated */
       if(mgk_uniq_annot(&orig, op))
          printf("Can change to xor by %d ^ %d\n",
                 orig, mgk_int_cst(op));
     }
   ;
}
```

5 Pattern Implementation

The biggest problem with implementing pattern matching is mapping the code specified in a pattern to the internal tree structure that the compiler translates its input source to. Doing this mapping for all possible pattern constructs is tedious at best; for a C++ compiler such as g++ it is completely out of the question. The difficulty of doing so caused our previous, non-language-based system [11] to force the user to manually construct the pattern matching AST trees on demand. Constructing such code requires a working knowledge of the compiler's tree structure. In our experience, having the language makes a breathtaking difference in the time, effort, and knowledge required to write an extension.

Our current system avoids most of the problem using a simple trick: the base compiler we are using is already able to map source code to its internal trees. Therefore, if we can map our patterns to source code and then run it on this code, it will generate the correct trees for us. Since patterns closely mirror input source, this translation is relatively straightforward. First, we generate a function that holds the given pattern as normal code, with all hole variables specified as parameters, and run the compiler on it. For example, the pattern:

```
decl { double } d;

// match calls to 'pow' with identical arguments
pat = { pow(d, d) };
```

it translated to the function:

```
int mgk_expr_pattern0(double d) {
     mgk_expr(pow(d, d));
}
```

After stripping off its function boilerplate, the produced tree will be (roughly) identical to that of the source code we want to match. We save this tree and match it against input functions by performing a recursive tree comparison. The call `mgk_expr` is used to tag the pattern as an expression; the alternative is to mark it as a statement. This marking is needed since the result of an expression may not be used (e.g., if it is a single variable reference), and thus it may need to be wrapped in a construct such as a function call to prevent the front end from eliminating it. For `g++`, `mgk_expr` is the identity macro.

This approach has two main challenges. First, there needs to be a way to specify matches on constructs that do not appear in the language ("any statement," "any function call," "any scalar type", etc). We counter this problem using special names (both in variables and in types) to convey the refinements we care about. Second, the same construct used in different contexts may generate different trees. The pattern matcher has to be aware of such synonyms. In practice, there are a small number of such cases; handling them in the matcher is not difficult, but is tedious.

At a high level, the matcher recurses down the input and pattern trees performing the following comparisons:

1. Identical nodes match;
2. If a pattern node is a parameter (hole), it matches if (1) its type is equivalent to the corresponding (sub)tree in the comparison tree and, (2) if it is used more than once in a pattern, all occurrences must match identical subtrees in the comparison tree.
3. If a pattern node is a special form or name, the matcher checks if its associated construct is present in the comparison tree (more below).
4. If a pattern subtree has a synonym that matches in the corresponding input tree using the above three steps.
5. All other nodes cause the comparison to fail.

5.1 Handling special forms

Metal is designed to be easy to integrate into existing compilers. A practical challenge to doing so is adding the numerous new types *metal* defines. While these names give clients cleaner, more precise control of matching, adding new first-class types to existing compilers can be difficult. To counter this, we have designed *metal* types so that they can be expressed in terms of existing ones. Most *metal* special types are mapped to C typedefs:

```
typedef unsigned any_scalar;
typedef void *   any_ptr;
```

We have modified the `g++` frontend to preserve these names (many C descended compilers evaporate them). During matching, if an input tree node matches a parameter that has one of these special types, we compare them.

This approach has the enormous practical benefit that it requires minimal front end modifications. As a result, adding new special types takes a matter of minutes. However, it creates two difficulties. First, the underlying overloaded type may result in different code in different contexts. For example, since the `any_scalar` type is really an `unsigned` type, if it is used in a floating-point context, the underlying tree will contain a code to do the promotion. The matcher handles this by skipping such artifacts. Second, the underlying type system may not provide a type that can stand in for everything. We currently lack a true "any" type, since C will not allow pointers and integers to be interchanged transparently. The front-end modifications required to turn this off during type checking are not difficult, but we have not done them.

As another example, representing an "any pointer" type as an actual type means that we must give it a size. If this size is used in calculations (e.g., indexing into an array) it will result in a different value. We explicitly correct for sizes in addressing calculations.

The pattern matching machinery is packaged so that it can be used outside of the language.

5.2 Handling macros and preprocessing

A nice side-effect of patterns mirroring source code is that they can be transparently used with macros. Rather than forcing clients to manually determine the code that the macro can expand to, they can just include the macro call in their pattern. In our previous system, specifying such code was tedious; the stylized way to handle it was to redefine the macros as functions or, in the case where the macro was doing syntactic transformations, to place function call "markers" inside the macro body. This had the twin disadvantage of (1) requiring source code modifications rather than working transparently and (2) making it difficult to run the resulting code in production use, since at the very least, macros were now replaced with more expensive and more environment-affecting functions.

```
/* Check for side-effects in the assert() macro */
sm Assert {
    decl { any } expr, x, y, z;
    decl { any_call } any_fcall;
    decl { any_args } args;

 start: { assert(expr); } ==>
           { mgk_recurse(expr, in_assert); }
      ;
 in_assert:
   { any_fcall(args) } ==> { err("function call"); }
        | { x = y }     ==> { err("assignment"); }
        | { ++z }       ==> { err("pre-increment"); }
        | { z++ }       ==> { err("post-increment"); }
        | { --z }       ==> { err("pre-decrement"); }
        | { z-- }       ==> { err("post-decrement"); }
        ;
}
```

Figure 1: A *metal* SM to catch side-effects in calls to the C `assert` macro. The function `mgk_recurse` will recursively apply the SM to the tree representing `expr` in state `in_assert`

An ancillary problem with macros is that the C preprocessor defines a variety of tokens that are replaced with contextually different values. Examples include `__FILE__` (replaced with the current file name) and `__LINE__` (replaced with the current line number). When these appear in patterns, they will be pre-bound to (most likely) values that will not appear in client code, giving spurious mis-matches. In addition, it also provides operators such as `#` to construct argument-specific tokens. To handle these cases, we modified the preprocessor so that while processing our pattern functions it all such uses with the special `all` keyword, which matches anything. This replacement allows extensions such as the `assert` checker in Figure 1, which looks for side effects in assertions, to transparently match all `assert` uses without having to violate modularity by redefining the macro itself. This approach has the additional benefit that checkers do not have to realize which of the constructs they match on could be macros.

Our approach will match raw C code that is identical to any macro expansion in a pattern. In most cases this is a feature, since the intention is to match syntactically equivalent constructs. It would not be difficult to handle the obverse case, where only macro invocations are matched, by marking the name of the macros at their invocation point. We have not needed such functionality, so have not added it.

6 Path-traversal Implementation

Conceptually, SM the analysis pass is implemented as a depth-first control flow graph traversal. The traverser is invoked after *xg++* parses every source function with a pointer to the AST for this function. It then constructs a simple control flow graph (CFG). Each "statement" in the CFG is a single AST node. The most straightforward analysis then applies the SM to this single CFG. While global analysis is supported by additional runtime libraries, the current version is not integrated into the language, and so clients have to use standard C routines to continue the analysis across function calls. In this paper we focus on our local analysis technique.

```
def analysis(sm m, block b, int state)
        for each s in nodes(b)
                if state in states(s)
                        return
                else
                        states(s) = states(s) U state
                        state = visit(m, s, state);

        if empty(succs(b))
                visit(m, NULL, state);
                return

        for each b' in succs(b)
                analysis(m, b', state)
```

Figure 2: Pseudo-code for our caching depth-first traversal algorithm.

In most cases when a pattern matches, the extension would like to skip visiting the subtrees and statements in the pattern. This skipping is implemented by having an implicit path variable called `mgk_skip`. When a pattern is matched, the end of the pattern is saved into `mgk_skip`, and the traversal function does not call the SM until after the end of the pattern. By making the skip cursor a path variable it is transparently saved and restored across different paths.

Given a CFG, our naive algorithm performs a simple depth-first traversal with backtracking. However, while this is sound, it is expensive because it is proportional to the (exponential) number of paths in the procedure. Thus, our system also implements a traversal algorithm that uses memoization to reduce the time to roughly linear in practice. The algorithm exploits the fact that many SMs are deterministic: given the same input they produce the same output. Thus, if a SM reaches a node in a path in the same state, with the same input, it must compute the same results and the entire path can be pruned.

Figure 2 shows pseudo-code for the memoization-based search algorithm. The function `visit(m,s,state)` invokes the SM `m` on the AST node `s` with the values of its path variables (including `mgk_state`) loaded from a global array indexed by `state`. The visit is recorded by adding `state` to the set `states(s)` stored in the node. If the SM later visits the same node again with the same `state`, the algorithm immediately backtracks. The algorithm assumes that the SM's behavior at a particular node is completely determined by the values of the path variables along with `mgk_state`. If this is not the case, the traversal must be done with the slower, non-caching implementation. We have written an extension to conservatively check for deterministic code in extensions but we do not mandate using it.

Note that if the SM has only one state, the analysis function degenerates into depth first search. SMs that examine particular nodes in the AST without considering the paths leading to them can simply define an initial state and never leave it.

6.1 Traversal time complexity

In the absence of path variables and raw assignments to `mgk_state`, the time complexity of the search algorithm in Figure 2 is approximately $O(S \times N)$ where S is the number of states in the SM description and N is the number of AST nodes in the function. The caching mechanism will ensure that each of the N nodes is visited at most S times; once a node has been visited in all S states, subsequent visits to that node will immediately backtrack instead of searching further. This bound is not quite correct because it assumes that the time to visit a node and perform a pattern match is $O(1)$. In reality the time varies with the size of the pattern and the frequency of certain types of tree nodes in the program being compiled. However, we have found that in all practical cases we have encountered, the matching time is small because most patterns are small. Longer patterns tend to match infrequently and are typically rejected after examining very few nodes. Future work will implement caching for the pattern matching to ensure that patterns matches are only attempted once per AST node, which will allow a tighter upper bound on the time complexity for the entire algorithm including the cost of pattern matching and searching.

With path variables and raw C assignments to `mgk_state`, the time complexity of the algorithm depends heavily on the behavior of the SM and must be determined by the extension writer if desired.

6.2 Saving and restoring path variables

The values of path variables are saved and restored along a path just as the `state` parameter in the algorithm in Figure 2. To support this behavior, we create a structure with fields for all of the path variables for each SM, including the implicit path variables `mgk_state` and `mgk_skip`. During the analysis we keep a set of these structures; the `state` parameter becomes an index into the set, and in the `visit` function the structure's fields are unpacked into local variables in the SM's scope. When the SM is finished visiting a node, the values of the path variables are packed back into a structure and inserted into the set, and the index is returned as the value of `visit`.

6.3 Per-variable state coupling

States are coupled to variables (or arbitrary AST nodes) by allocating a state variable and a path variable of internal AST type `tree` for every hole variable used in a `var.state` expression. The state variable stores the state associated with the tree variable along the current path. The tree variable stores the AST node within a pattern that matches the hole `var`. When a transition is guarded by a `var.state` expression, the transition will only trigger if the `var` in the pattern matches one of the saved `tree` nodes. Any state changes for `var` affect the state variable associated with the `tree` node that matches.

FLASH	**Errors**	**False Pos**
Message length consistency	17	2
XOR opcode checker	2	10
Linux	**Errors**	**False Pos**
Locking conventions	9	102
Exokernel	**Errors**	**False Pos**
Assert side effects	~70	~35

Table 2: A sampling of the checkers we have implemented using *metal* .

FLASH	**Optimizations**	**False Pos**
Buffer free	11	9
XOR opcode optimization	hundreds	~10

Table 3: A sampling of the optimizers we have implemented using *metal* .

7 Discussion

Metal makes checkers and optimizers easy to write. We have built numerous *metal* extensions have found many bugs in real systems, most of which are less than a hundred lines of code and can be written in a few hours. In this section we discuss some of the checkers we have written, the results obtained from applying them, and our experience

7.1 Applications

Table 2 gives a sampling of the checkers we have written using *metal* and the number of errors and false positives they found. Table 3 gives a few of the extensions we have written to look for opportunities for optimization. A comprehensive examination of these results can be found in a pair of papers on the use of meta-level compilation [4, 8]; here we give a brief description of each extension.

1. The FLASH machine's programmable node controllers impose many systems-type restrictions on the protocol handlers running on the nodes [14]. We wrote several checkers to check these restrictions and look for potential optimizations. **Message length consistency:** when a handler sends a message the length of the message is in a global header field while the presence of a message body is indicated with a parameter to the message send function. Our extension checks that the possible values of the length field are consistent with with the parameters to the send function. **XOR opcode checker:** we ensured that when XOR was used to change message opcodes that the resulting opcode was valid for the type of message. **Buffer free optimization:** buffers may be freed sooner in some cases with a two line change; we check for this condition. **XOR opcode optimization:** As described in Section 4, opcode assignments can often be replaced with XOR, saving one instruction.

2. Linux device drivers are often written by programmers without a clear understanding of the locking subsystem. We wrote checkers for this subsystem and found several potential errors in device drivers and networking code; we have not yet been able to confirm most of these potential bugs with the authors of the code. **Locking conventions:** we check the stylized kernel convention of acquiring and releasing locks within the same function. We also conservatively check that functions do not exit with interrupts or "bottom halves" disabled and that locks are not doubly locked or unlocked. The interrupt checker described in sections 2 and 4 is a stripped down version of this checker.

3. The Exokernel is an experimental research operating system that presents a low-level machine interface to applications [13]. **Assert side effects:** Many of the assertions in the Exokernel code contain expressions with side effects; when the assertions are removed the side-effects disappear. Figure 1 shows the checker we wrote to conservatively warn about possible side-effects.

Our experience with writing checkers for these systems shows that *metal* is a practical tool for weeding out errors and finding opportunities for optimization. The conservative static nature of the checkers manifests itself in the number of false positives produced and probably a number of silent false negatives. The number of false positives in some of the checkers could be reduced by writing more sophisticated extensions. For example, some of the false positives produced by the Linux locking checker were caused by high-level locking functions that used the primitive locking functions; to our checker these functions look like errors because they exit while holding a lock. The checker could be made more sophisticated by adding a knowledge of non-primitive locking functions.

7.2 Experience

The *metal* language is still relatively new, and most of *metal* programs are relatively short. Yet these extensions are powerful for their size, and the following features help explain the economy.

Patterns. We found patterns to be a huge improvement over manually traversing and matching AST nodes. Many of our old extensions had a complex set of routines to examine AST nodes and a relatively simple function to implement the SM. When we converted these extensions to *metal* , virtually all of the AST examination routines disappeared. Rather than having to understand complex g++ internals, patterns allowed us to write C code to specify constructs in the language in an intuitive and flexible way.

This simplicity can greatly reduce extension complexity. Our experience when rewriting the full-fledged version of the FLASH xor checker discussed in Section 4 is typical. The original non-*metal* version was roughly 500 lines of code, and required an intimate knowledge of how *xg++* tree's were implemented. The *metal* version of this checker was roughly 200 lines of code, provided more sophisticated error reporting, and is much higher-level. Additionally, the checker catches a wider range of errors. This has happened numerous times when translating checkers to *metal*— the simplicity of the language quickly points out holes hidden in the original versions.

Per-variable state. We implemented the Linux locking convention checker three times: once before implementing *metal* , once after implementing *metal* with path variables but before per-variable state, and finally with all of the features described in this paper. The original checker essentially implemented path variables manually and took about 500 lines of C code; the second implementation took advantage of built-in path variables and took about 300 lines of *metal* ; and the final version used per-variable state to whittle the checker down to about 100 lines of *metal* .

The bad. Our mistaken assumptions about the complex internal AST structure bit us several times while implementing *metal* . In one instance, the g++ backend modified the trees for some "asm" statements (inline assembly) after being examined by our frontend hook. Since *xg++* works one function at a time, patterns containing asm statements got mangled but the trees they were matched against were examined before being mangled. We fixed these implementation artifacts by changing the pattern matcher to explicitly take them into account.

Pattern synonyms pose two difficulties for the extension writer. First, it is easy to overlook synonyms when writing patterns because missing one might only cause false negatives, thus silently failing to find potential bugs. Second, synonyms expose artifacts of the underlying AST structure that the extension writer would prefer to ignore. For example, should the pattern " p = malloc(sz) " match "void *p = malloc(sz)"? To most extensions these constructs are essentially identical, but in g++ they are represented by different types of tree nodes – the former is an assignment while the latter is a variable declaration with an initializer. Synonyms can be handled by putting a special case into the pattern matcher, but it is not clear that this is always desirable. We are investigating methods to resolve this problem by giving the extension writer an interface with sensible defaults and fine-grained control over the matching process.

Finally, incompatibilities between C and C++ and their implementations in gcc and g++ have caused us a fair amount of pain. The code in large C-based systems such as Linux often depend on constructs that are C but are not C++ [17]. Worse, Linux uses GNU-specific extensions to gcc that are not supported (and sometimes won't even parse) in g++. To solve this problem we have done everything from relaxing g++'s strict typing conventions to modifying macro definitions and changing the program source code.

7.3 Future Work

Other languages. We intend to both port *metal*'s runtime system from *xg++* to other compilers and to integrate *metal* with other languages such as Java. Porting the backend would show that *metal* extensions can be portable across different compilers. Also, porting to a stable C compiler would allow us to avoid some of the language compatibility problems between C and C++. Integrating into other languages would allow us to explore the language-independent idea of meta-level compilation in other domains.

Soundness. *Metal* checkers tend to be simple static compiler analysis passes. Thus, depending on the check, they can be overly conservative and even unsound. While they have provided effective support for finding errors in many situations, we want to extend them with more powerful techniques so that they can scale up to verifying various correctness properties. Two obvious candidates are incorporating state of the art pointer analysis and abstract interpretation algorithms. Other promising methods are integrating extensions with existing model checking or theorem proving systems.

8 Related Work

We proposed an open compiler model to support Meta-level Compilation (MC) in [10, 11] and provided a simple system, `magik` (based on the `lcc` ANSI C compiler), for using it. Later, we implemented MC in *xg++* and gained experience checking systems such as FLASH [8] and Linux [4]. *Metal* was developed after we converged on the SM model and realized how patterns made a language feasible. While the earlier papers focused on the applications of MC, this paper demonstrates a language and an implementation that greatly reduces the amount of expertise needed to develop extensions. Below, we compare our work to macro systems and high-level compilation, and other program analyses.

Macro systems are the most venerable method for user-controlled language extension. An advantage of such systems (e.g., Lisp) is their tight integration with the source language. One difference between macro systems and *metal* is the intent: macros exist intermingled with the code and transform it at the syntax level, whereas *metal* extensions exist largely independent of the source base and are meant to examine code at a somewhat higher level with access to flow information and compiler data structures. Macros are not usually used to statically check code, whereas *metal* was designed to make expressing static checks easy. Weise and Crew's recent work with programmable macros [19] uses a pattern-like technique of using an extension of the base language to ease the expression of code fragments. One major difference is that patterns are meant to match free-standing code whereas macro invocations are explicit. Macros perform substitution but not pattern matching.

Crew's ASTLOG [9] is a Prolog-based tool for traversing abstract syntax trees. ASTLOG uses a set of primitives to manually inspect tree structures; it has neither a pattern-like facility to simplify matching nor a way to access the control flow information needed to perform path-based analysis.

Compile-time Meta-Object Protocols (MOPs) such as Shigeru Chiba's `Open C++` [7, 6] expose a compiler's data structures in an object-oriented manner to allow compile-time "meta objects" to control the compilation process. `Open C++` can specialize matrix arithmetic, provide persistent objects, and perform limited lexical extension. `Open C++`, like other MOPs, tends to focus on localized transformations in an object-oriented framework without much access to control flow information.

Many groups have developed program analysis frameworks [1, 3, 12, 18, 2]. These projects have focused on making program analysis efficient at the back end, but their interfaces typically require the specification of a relatively complex mathematical model of the analysis.

Work on user-extensible dataflow analysis in FLAVERS [15] and Cesar [16] bear the most similarity with our back-end analysis. Both systems use flow analysis with a property-specifying finite state machines to check for sequencing and other violations. These systems seem to lack a pattern facility; the systems emphasize the power and performance of the analysis technique.

Parser generator tools such as yacc and bison provided a syntactical model for *metal* because they share the common feature of base-language escapes. The syntax of combining rules was also derived from these tools.

9 Conclusion

Metal extensions are simple and powerful. Their simplicity allows non-compiler writers to readily implement extension passes. Their power allows them to catch serious errors in systems ranging from Linux to FLASH to the Exok exokernel. We believe these results demonstrates *metal* can serve as an effective basis for finding errors and optimization opportunities in a variety of domains.

References

[1] A. Aiken, M. Fahndrich, J.S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Second International Workshop on Types in Compilation*, Kyoto, Japan, 1998.

[2] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with pag. In *SAS'95, Static Analysis*, pages 33–50, 1995.

[3] C. Chambers, J. Dean, and D. Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report 96-11-02, University of Washington, November 1996.

[4] B. Chelf, A. Chou, S. Hallem, and D.R. Engler. Systems as languages. Submitted for publication, available on request: bchelf@cs.stanford.edu, April 2000.

[5] B. Chelf, W. Huang, Yu-Ping, S. Lindholm, and D.R. Engler. Meta-level optimization. Submitted for publication, available on request: bchelf@cs.stanford.edu, April 2000.

[6] S. Chiba. Open C++ programmer's guide. Technical Report TR93-93-3, Dept. of information science, University of Tokyo, 1993.

[7] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA 1995 Conference Proceedings Object-oriented programming systems, languages, and applications*, pages 285–299, October 1995.

[8] A. Chou, B. Chelf, D.R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. Submitted for publication, available on request: acc@cs.stanford.edu, April 2000.

[9] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the First Conference on Domain Specific Languages*, pages 229–242, October 1997.

[10] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the First Conference on Domain Specific Languages*, October 1997.

[11] Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, May/June 1999.

[12] M.W. Hall, J.M. Mellor-Crummey, A. Carle, and R. Rodriguez. Fiat: A framework for interprocedural analysis and transformation. In *The Sixth Annual Workshop on Parallel Languages and Compilers*, August 1993.

[13] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[14] J. Kuskin, D. Ofelt, M. Heinrich, John Heinlein Richard Simoni, Kourosh Gharachorloo, John Chapin David Nakahira, Joel Baxter, Mark Horowitz Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[15] Gleb Naumovich, Lori A. Clarke, Leon J. Osterweil, and Matthew B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of the 19th International Conference on Software Engineering*, pages 594–595. ACM Press, 1997.

[16] K. Olender and L. Osterweil. Cesar: A static sequencing constraint analyzer. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 66–74, December 1989.

[17] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, 1986.

[18] S. Tjiang and J. Hennessy. Sharlit – a tool for building optimizers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 82–93, San Francisco, CA USA, June 1992. ACM Press. Published as SIGPLAN notices, vol. 27, no. 7.

[19] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of PLDI '93*, pages 156–165, Albuquerque, NM, June 1993.