Pluggable Policies for C *

Mark Hills Feng Chen
Grigore Roşu
University of Illinois at Urbana-Champaign
{mhills,fengchen,grosu}@cs.uiuc.edu

January 23, 2008

Abstract

Many programs make implicit assumptions about data. Common assumptions include whether a variable has been initialized or can only contain non-null references. Domain-specific examples are also common, with many scientific programs manipulating values with implicit units of measurement. However, in languages like C, there is no language facility for representing these assumptions, making violations of these implicit program policies challenging to detect. In this paper, we present a framework for pluggable policies for the C language. The core of the framework is a shared rewrite-logic semantics of C designed for symbolic execution of C programs, and an annotation engine allowing for annotations across multiple policies to be added to C programs. Policies are created by providing a policy specific language, usable in annotations, and policy specific values and semantics. This provides a method to quickly develop new policies, taking advantage of the existing framework components. To illustrate the use of the framework, two case studies in policy development are presented: a basic null pointer analysis, and a more comprehensive analysis to verify unit of measurement safety.

1 Introduction

Many programs make implicit assumptions about data. Common examples across many languages include assumptions about whether variables have been initialized or can only contain non-null references. Domain-specific examples are also common; a compelling example is units of measurement, used in many scientific computing applications, where different variables and values are assumed to have specific units at specific times/along specific execution paths. These implicit assumptions give rise to implicit domain policies, such as requiring assignments to non-null pointers to also be non-null, or requiring two operands in an addition operation to have compatible units of measurement.

*Supported by NSF CCF-0448501 and NSF CNS-0509321.

In languages like C, there is no language facility for representing these assumptions, making violations of these implicit program policies challenging to detect. For instance, in the following code example, assuming that `x` and `y` both have associated units of measurement, it may not be obvious whether the returned value is unit correct (it is, regardless of the intended units of `x` and `y`).

```
double f(double x, double y) { return x * y; }
double g(double x, double y) { return x / y; }
double h(void) {
  double x = 2.5, y = 3.5;
  return x + g(f(x,y),y);
}
```

Although this can be visually verified, one can imagine much larger programs, with many more calculations, where this would not be possible. Because of problems such as this, a number of techniques have been developed to check implicit program assumptions (see Section 6). These include techniques making use of type system extensions, such as type qualifiers[12], often including a form of type inference; techniques based on modifying language constructs and/or language semantics; and techniques based on programmer-added annotations. While some techniques have attempted to be generic, most have not, but have instead focused on specific domain areas, like pointer policies or unit safety.

In this paper, we present a framework for *pluggable policies* for the C language which allows these implicit policies to be made explicit and checked. The core of the framework is a shared annotation engine and parser, allowing annotations in multiple policies to be inserted by developers as comments in C programs, and a shared rewrite-logic semantics of C, focused on symbolic execution, designed as a number of reusable modules that allow for new policies to be quickly developed and plugged in (for instance, the case study in Section 4 was developed in under two days). Background on term rewriting and rewriting logic is presented briefly in Section 2, while the framework, which uses rewriting logic and term rewriting techniques, is described in depth in Section 3. To illustrate the use of the framework, two pluggable policies are presented as case studies: a simple pointer analysis, in Section 4, and a comprehensive units of measurement safety policy, in Section 5. These policies should provide a starting point for the development of other custom policies. Related work, with a focus on C program analysis and unit safety, is discussed in Section 6, with conclusions and potential future work presented in Section 7.

2 Term Rewriting and Rewriting Logic

This section provides a brief introduction to term rewriting, rewriting logic, and rewriting logic semantics. Term rewriting is a standard computational model supported by many systems; rewriting logic [26, 25] organizes term rewriting modulo equations as a complete logic and serves as a foundation for programming language semantics and analysis [27, 28].

2.1 Term Rewriting

Term rewriting is a method of computation that works by progressively changing (rewriting) a term. This rewriting process is defined by a number of rules – potentially containing variables – which are each of the form: $l \rightarrow r$. A rule can apply to the entire term being rewritten or to a subterm of the term. First, a match within the current term is found. This is done by finding a substitution, θ, from variables to terms such that the left-hand side of the rule, l, matches part or all of the current term when the variables in l are replaced according to the substitution. The matched subterm is then replaced by the result of applying the substitution to the right-hand side of the rule, r. Thus, the part of the current term matching $\theta(l)$ is replaced by $\theta(r)$. The rewriting process continues as long as it is possible to find a subterm, rule, and substitution such that $\theta(l)$ matches the subterm. When no matching subterms are found, the rewriting process terminates, with the final term being the result of the computation. Rewriting, like other methods of computation, can continue forever.

There exist a large number of term rewriting engines, including ASF [33], Elan [5], Maude [9, 10], OBJ [15], Stratego [34], and others. Rewriting is also a fundamental part of existing languages and theorem provers. Term rewriting is inherently parallel, since non-overlapping parts of a term can be rewritten at the same time, and thus fits well with current trends in architecture and systems.

2.2 Rewriting Logic

Rewriting logic is a computational logic built upon equational logic which provides support for concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined. The equations specify which terms are considered to be equal. All equal terms can then be seen as members of the same equivalence class of terms, a concept similar to that from the λ calculus with equivalence classes based on α and β equivalence. Rewriting logic also provides *rules*, which represent concurrency; we use only the equational subset of rewriting logic here, so we focus just on equations, and use the terms "equation" and "rule" interchangeably, instead of in their more technical sense. Rewriting logic is connected to term rewriting in that all the equations, of the form $l = r$, can be transformed into term rewriting rules by orienting them properly (necessary because equations can be used for deduction in either direction), transforming them into $l \rightarrow r$. This provides a means of taking a definition in rewriting logic and a term and "executing" it. While the definition can be the standard evaluation semantics of a language, it can also be an abstract semantics using domain-specific values, such as types or units. In this paper, "evaluation" or "execution" refers to symbolically evaluating a program, with expressions and statements manipulating abstract values and program states.

```
eq exp(E + E') = exp(E,E') -> + .
eq k(lookup(X) -> K) env(Env [X,L,V]) =
   k(val(lvp(L,V)) -> K) env(Env [X,L,V]) .
```

Figure 1: Sample C Semantic Rules, in Maude

2.3 Rewriting Logic Semantics

The semantics of C is defined using Maude [9, 10], a high-performance language and engine for rewriting logic. The current program is represented as a "soup" (multiset) of nested terms representing the current computation, environment (mapping names to locations in memory or directly to values), analysis results, bookkeeping information, and analysis-specific information. The most important piece of information is the `Computation`, named `k`, which is a first-order representation of the current computation, made up of a list of instructions separated by `->`. The `Computation` can be seen as a stack, with the current instruction at the left and the remainder of the computation to the right. This methodology is described in more detail in papers about the rewriting logic semantics project [27, 28].

Figure 1 shows examples of Maude equations included in the C semantics presented in Section 3. The first shows an example of an equation used to take apart an expression; when `E + E'` is encountered, both `E` and `E'` need to be evaluated first, with the operation (here `+`) put on the computation to "remember" which operation was being performed. This can be seen as being similar to a stack machine, where the operation and expressions are placed on the stack, with the intention being that the operation can continue once the expressions are reduced to values. The second represents a memory lookup operation. Here, if identifier `X` is being looked up, and the environment set contains a set item with the name, some location `L`, and some value `V`, a location value pair `lvp` containing the location and the value, `lvp(L,V)`, is returned in place of the lookup operation, while the environment remains unchanged.

3 A Framework for Pluggable Policies in C

In this section we present a framework for pluggable policies in C. This framework can be viewed both from the perspective of a C programmer and a policy creator. To the C programmer, this framework is exposed through policy-specific annotations made in the source code in comments, requiring no change to the underlying language. These annotations are used to indicate function preconditions and postconditions, as well as assertions and assumptions inside function bodies. An annotation processor and parser are then used to process the annotated source files and generate a formal program representation, which is checked in a policy determined by the programmer, with output either indicating that the program has passed the checker or listing a series of warnings

```
1  //@ pre(UNITS): unit(w) = lb
2  //@ post(UNITS): unit(@result) = kg
3  double lb2kg(double w) {
4    double rv = 10 * w / 22;
5    /*@ assume(UNITS): unit(rv) = kg */
6    return rv;
7  }
```

Figure 2: C Code, with Annotations

and errors by program line.

From the perspective of the policy creator, the framework consists of the pieces mentioned above, but also includes a general-purpose semantics of C in Maude and a number of policy-specific extensions. Each policy can make use of its own custom notation, can provide its own custom rules to determine correctness, and can issue its own custom error messages. Each policy can also leverage existing parts of the framework, allowing most of the core framework code to be reused and providing a powerful environment for adding and extending policies.

3.1 Code Annotations

Each policy provides a policy-specific language for specifying program requirements and assumptions. This language is used by the programmer when writing annotation comments, which are standard C comments but which start either with `/*@` (for block comments) or `//@` (for line comments). Preconditions and postconditions are allowed before the start of a function, with zero or more conditions allowed for each policy (i.e., it is legal both to include multiple preconditions for the same policy and to include preconditions for different policies, both on the same function). Annotations inside function bodies are for assertions and assumptions, again allowing for multiple of each across multiple policies. Some policies, such as that presented in Section 4, need few annotations (individual programs may have none), but still provide an annotation language for cases where the programmer wants to assume or assert that certain domain facts hold. An annotation example is shown in Figure 2. This figure shows a function, `lb2kg`, for converting double values from pounds to kilograms; the policy is that presented in Section 5. The precondition states that the input parameter, `w`, has a unit of `lb`, while the result of the function, indicated as `@result`, is guaranteed to be `kg` by the postcondition. According to the rules for manipulating units, the variable `rv` will be assigned the unit `lb`, since this is the unit assigned to `w`, and since constants effectively have no unit; the assumption in the function body indicates that `rv`, from the point of the assumption forward, should instead be treated as having unit `kg`. This assumption is how a programmer represents the conversion to a new unit, making sure the postcondition holds.

To allow multiple policies to be used in a single program, each annotation

specifies the domain being checked by including the name of the policy; for instance, in Figure 2 the *policy tag* on the annotations is `UNITS`. Each policy knows which tags are associated with it, and will only check annotations which include the proper policy tag(s). While it is possible to leave out the policy tag and specify a default tag instead, this is only appropriate in cases where only one policy will ever be used.

3.2 Parsing C

Once the source code has been annotated, it needs to be parsed to allow the annotations to be read in and verification tasks (chunks of code to be verified) to be generated. Parsing takes place in two phases. In the first, a simple transformation is applied to move annotations from comments into language syntax, producing a program in a version of C, used internally by the framework, that has been extended with assertions, assumptions, and function preconditions and postconditions. This stage occurs before preprocessing, and is written to ensure that line numbers are not modified so errors can be accurately reported during policy checking.

The second phase takes this modified version of the source, preprocesses it using a standard C preprocessor (such as `gcc -E`), then parses the preprocessed source using a modified version of the CIL parser for C [29], generating an internal, AST-like representation of the code. This internal representation has also been extended to be aware of assertions, assumptions, preconditions, and postconditions, but does not know about the policy languages used in the annotations, allowing the parser to remain policy generic (but also meaning that syntactic errors in the annotation formulae are not caught during parsing).

Once in this form, the various analysis passes written for CIL are available for use on the source code. These have been augmented with several additional passes which prepare the code for policy verification. To ensure that verification tasks do not grow too large, instead of generating one task for the entire program, one is generated for each function. To do this, a custom pass modifies the function bodies, with each call site in a function body replaced by assertions and assumptions based on the preconditions and postconditions specified for the called function – assertions to guarantee preconditions hold at the time of the call, assumptions to specify what information can now be assumed after the call. Formal parameters used in the preconditions and postconditions are replaced by the actual parameters used at the time of the call. A simplifier, based on the provided CIL simplifier, ensures that expressions are in three-address form, allowing simpler expressions (generally variable names) to be used as actuals during this replacement. Along with this precondition and postcondition inlining process, the preconditions of a function are made available as assumptions at the start of the function body, while postconditions are checked by adding assertions before each return instruction.

Once this process has been completed, and each function can be viewed as an independent verification task, a modified CIL printer class is used to generate each verification task, producing code similar to standard C source code but

```
1  void example(void) {
2    double x = 3.5; //@ assume(UNITS): unit(x) = lb
3    double y = lb2kg(x); //@ assert(UNITS): unit(y) = kg
4  }
```

Figure 3: C Code, with Function Call

```
1  double x;
2  double y;
3  double tmp;
4  {
5    x = 3.5;
6    #Fassume(UNITS,unit(x) = lb);
7    #Fassert(UNITS,unit(x) = lb);
8    #Fassume(UNITS,unit(tmp) = kg);
9    y = tmp;
10   #Fassert(UNITS,unit(y) = kg);
11   return;
12 }
```

Figure 4: Generated C Code for Verification

with some modifications to ease subsequent Maude parsing. This produced code includes line number information, used by the policy checkers to produce useful error messages, and includes the capability to also pass in additional information, such as declared types, global variables, and analysis information more easily performed on the source code (such as alias information computed on the entire program, which would not be available when looking at just a single function body).

An example of this process of verification task generation is illustrated in Figures 3 and 4. Figure 3 shows a simple function in C that declares two variables, `x` and `y`, of type double. `x` is assumed to have unit `lb`, while `y` is asserted to have unit `kg` after the call to `lb2kg`, the code for which was shown above in Figure 2. Figure 4 then shows the code generated by the modified CIL parser (with line directives removed to reduce clutter) to be processed by the policy checker in Maude. One obvious difference is that the call to function `lb2kg` is gone, replaced with an assertion (based on the precondition) that the parameter passed to it, `x`, has unit `lb`, and an assumption (based on the postcondition) that the return value, assigned to `tmp`, has unit `kg`. Another difference is that a new variable, `tmp`, has been introduced as part of the three-address transform. Finally, the original `assume` and `assert` annotations present in function `example` have been transformed, in place, into `assume` and `assert` directives, written respectively as `#Fassume` and `#Fassert`. Also, note that the function header is gone; any function parameters will instead be listed as

```
op _+_ : Exp Exp -> Exp .
op if__else_ : Exp Stmt Stmt -> Stmt .
op return_; : Exp -> Stmt .
```

Figure 5: C Abstract Syntax

declarations before the function locals.

3.3 The Policy Framework

The policy framework is made up of the annotation and parsing components, described above, and an executable rewriting-logic semantics of C written using the Maude system. This semantics is made up of a number of modules, including multiple modules for abstract syntax, the semantic configuration (i.e., state), symbolic execution semantics, and policy semantics. Here we concentrate on the shared components of the framework; case studies illustrating pluggable policies are discussed in Sections 4 and 5.

Abstract Syntax An abstract syntax of C is defined as part of the policy framework to allow rules to be written over a syntax as close to native C syntax as possible. Syntactic categories, such as expressions and statements, are defined as rewriting logic *sorts*, with *operations* defined for each syntactic construct. These operations look very similar to BNF definitions, and are intended to have a familiar look and feel.

Figure 5 shows several examples of the C abstract syntax defined in the policy framework. The first operation defines the `+` operator as taking two expressions, one in each underscore, and yielding an expression; the second and third similarly define the `if` and `return` statements. These can be mentally converted to BNF by inserting the sorts in place of the underscores and moving the sort after the arrow to the front: `op if__else_ : Exp Stmt Stmt -> Stmt` then becomes `Stmt ::= if Exp Stmt else Stmt`.

Core Semantics While some of the semantics need to be tailored for specific policies, a complete set of core modules is provided as a basis for these policy extensions. Part of this definition is a generic *configuration*, or state, used by the core rules. The configuration is used to keep track of information needed by or produced during policy checking; Figure 6 shows a graphic representation of the configuration, with the type of information shown in the box and named lines showing the name used to access the information in the state. The core configuration includes several pieces of information, including: the computation, `k`; the current environment, `env`; a set of all environments, `envs`, the need for which is explained below; `out`, containing output messages generated during policy checking; `nextLoc`, holding the next (symbolic) memory location;

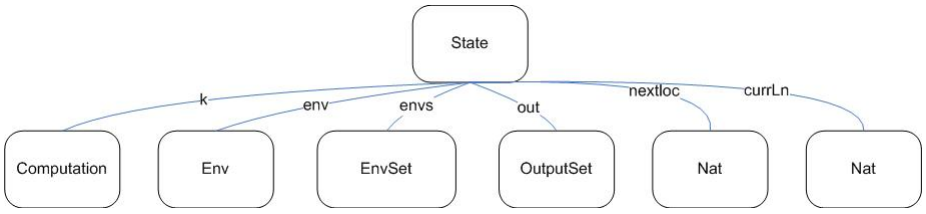

Figure 6: Framework State Infrastructure

and `currLn`, the current line number from the original source program. As checking progresses, rules will change the information in one or more of these configuration items.

The core semantic rules then fall into several categories:

- generic rules designed to manage the configuration or provide often-needed functionality, such as rules to issue warning messages;
- rules for expressions, which in core act just to start expression evaluation but do not reassemble values, the process for which is generally policy specific;
- rules for statements, which manage the environment and handle any environment splitting required to model different control-flow paths (like for conditionals);
- general rules used across multiple policy languages, such as the definition of policy expressions and some constructs used in many different policy languages (including standard logical operations, like conjunction or disjunction).

The most complex are the rules for statements. Most statements take and produce a single environment. Some can instead generate sets of environments, such as a conditional, where different environments can be generated when the condition is true and the then branch is entered, or when the condition is false and either the else branch is entered or, in cases where no else is present, the if body is not executed. Since conditionals can be nested, each branch can actually generate multiple environments. Other constructs that can generate environment sets include loops, switch statements, and gotos. Fortunately, CIL transforms the ternary expression into a conditional, allowing us to assume that expressions can never split environments.

To handle statements properly, two core operations are defined. The first, designated as just `stmt`, takes a statement and sets it up to execute in each environment in the current set of environments. This execution uses the second operation, `stmt!`, which `stmt` will invoke once for each environment in the environment set, merging back in the potentially altered environment(s) when the statement evaluation finishes. Policy-specific hooks are provided to properly merge in information after each statement evaluation, since different policies

may extend the core state in different ways, leading to different merging strategies. Also, to handle situations where statements need non-default processing, two additional operations, `isDirective` and `isSpecial`, are defined, with `stmt` only evaluating its statement when it is not marked as special or a directive. The intended meaning is that directives cannot modify the environment, such as with `#line` directives which just change the current line, while special statements may need additional setup beyond that provided by default.

Although the infrastructure discussed thus far provides support for statements that execute once, additional support is needed for statements inside loops (and for gotos, not discussed here). Since loop statements can repeat, it is possible for them to continually change the environment, generating new additions to the environment set with each iteration. To handle this, a counter is used for each loop. If, at the end of an iteration, the generated environment set contains no new environments, loop processing stops. If new environments are generated, the loop is processed again, with the counter incremented. If the counter threshold is reached without the environment set stabilizing, a warning message is issued, and processing continues for the rest of the function body. This allows for common cases to be handled, such as when the domain values stabilize in one or two iterations, while preventing non-termination of the analysis.

Sets of environments are not the only alternative for handling cases where different values are produced along different control-flow paths. One could also associate a set of values with each object/location, or use values such as $\top$ and $\bot$ to represent undefined and overdefined/error values. The advantage of environment sets is that they allow for increased precision, reducing false positives. For instance, in the following code, having sets of values or $\top$ and $\bot$ would indicate an error where there is none.

```
int x,y,z; //@assume(UNITS): unit(x) = unit(y) = m
if (b) {
  x = y = 3; //@assume(UNITS): unit(x) = unit(y) = f
}
z = x + y;
```

Using sets of values, the addition of `x` and `y` would raise an error, since both `x` and `y` could have unit `m` or `f`, and the addition would be invalid when `x` has one and `y` has the other. Using $\top$ and $\bot$, both `x` and `y` would be assigned $\bot$, since they would each possibly be assigned two incompatible units; summing them would again yield $\bot$. However, the addition is actually unit-safe, since `x` and `y` will have the same unit when summed, with both either having `m` or `f`.

The cost of using environment sets is a potential degradation in performance, especially in domains (like units) where there is a large, or potentially infinite, set of values. Here, some pathological programs can cause exponential increases in the size of the environment set. In many domains abstract values rarely split in this way, especially repeatedly, so this is not often a problem in practice. One potential solution is to set a high-water mark for the size of the environment

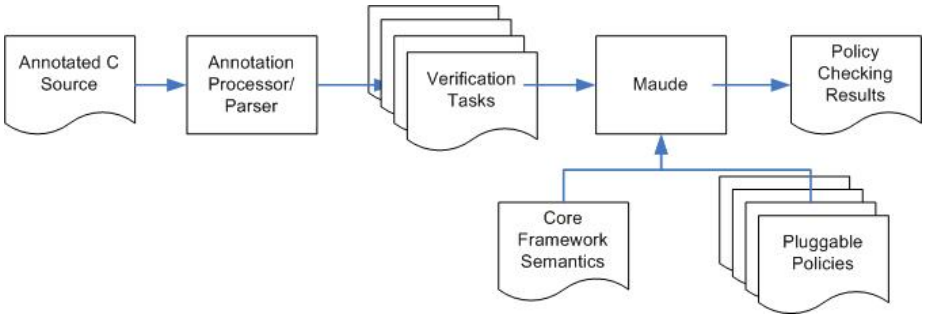

Figure 7: Framework Execution Model

set, discarding excess environments and issuing warnings when this mark is surpassed, which would allow policy checking to continue while alerting the user to a potential problem. Policies also can define their own notion of values, allowing an individual policy to opt for representations such as sets of values and avoid generating multiple environments.

Policy Checking The general policy checking process is shown in Figure 7. First, an annotated source file is processed by the annotation processor and parser; this process generates multiple verification tasks, one for each function in the source file. These verification tasks, along with the core semantics and the policies, are then handed to Maude; the core semantics and the policies are all definitions in rewriting logic, while the verification tasks are terms which are symbolically executed using term rewriting techniques. When this process terminates, any results output during policy checking are displayed to the user. These results are leveled, and can be filtered appropriately, allowing a policy to define a severity for each issued warning. For instance, a policy for units may define an attempt to add two values with incompatible units as an error with severity 1, but could define environment splits caused by conditionals as an informational message with severity 3, reflecting that splits may not cause errors but may (or may not) indicate a misunderstanding by the programmer.

4 Case Study: A Simple Pointer Policy

One common problem in many C programs is the accidental dereferencing of null pointers. This is part of a class of memory management problems caused by the explicit memory management model of C, with other common examples being leaking allocated memory, freeing the same chunk of memory multiple times, and accessing deallocated (and potentially reallocated) memory through dangling pointers. As an example of writing a simple policy using the policy framework, this section presents a policy for checking that null pointers are not dereferenced, including a policy language for specifying the "nullness" of pointers passed as function parameters and returned as function results. This policy is kept simple

Value	V ::=	`undefined` | `defined` VT
ValueType	VT ::=	`zero` | `other` | `ptr` L
Location	L ::=	`null` | `notnull` N

Figure 8: Not Null Policy: Policy Domain

for the purpose of this paper, and can be seen as a debugging policy, not a verification, since we don't attempt to catch all potential errors. An extended policy, intended to check for a wider class of errors (like dereferencing dangling pointers) and handle more complex memory usage scenarios (like pointers to offsets within structures) is available for download[4].

The first step to devising a new policy is to determine the policy's *domain* – the types of values that we will be using in the policy. Here, taking a lead from another paper on detecting C memory access violations [11], we divide values into two general categories: `defined` values, and `undefined` values. Within defined values, we keep track of `zero` (useful for null assignments and tests) and `other` for non-pointer values, and `ptr` for pointer values. The `ptr` value holds a *location*; for live pointers, other than standard symbolic locations this can also be `null`, representing cases where the pointer may possibly be assigned to `NULL`. Dead (deallocated) pointers are not included in the version of the policy presented here. The domain for this policy is shown in Figure 8, with N representing natural numbers (used as keys for symbolic locations in the environment).

Once these values are defined, the next step is to define the behavior of C programs over these abstract values. Most of this occurs at the level of expressions, since the expressions will manipulate the domain values directly. The most interesting expressions are those used for pointer dereference; while most expressions just move values around in a policy- appropriate manner, dereferencing can actually trigger an error. The rules for handling dereferencing are shown in Figure 9[1]

The first rule is a generic rule used by all policies, and simply states that, to determine the value of `* E`, first evaluate `E`, which will evaluate to an abstract value. Once it does, `*addr` is a reminder to check this value in light of its use in a dereferencing expression. The second rule represents the case where `E` evaluates to a null pointer: `lvp` is used to represent a "location value pair", which can be seen as the location where the object is kept in memory (`L`) and the value of the object in that location (`defined(pointer(null))`). This case represents an error, since it means the program will try to dereference a null

[1]This actually does not include all dereferencing rules, just the core rules used for not null checking, since other rules also ensure the pointer is defined – i.e., has had `NULL` or an address assigned to it at some point.

```
eq exp(* E) = exp(E) -> *addr .

eq k(val(lvp(L,defined(pointer(null)))) -> *addr -> K)
 = k(issueWarning(1,"Attempt to dereference a null pointer")
     -> val(lvp(noloc,undefined)) -> K) .

eq k(val(lvp(L,defined(pointer(L')))) -> *addr -> K)
 = k(llookup(L') -> K) .
```

Figure 9: Not Null Policy: Pointer Dereferencing

```
int *p = NULL;
if (p) {
  ... *p = 5; ... /* Should not cause a warning */
}
*p = 10; /* Should cause a warning */
```

Figure 10: Not Null Policy: Avoiding False Positives

pointer. `issueWarning` issues an error message, which will be tagged with the line number in the source program. Since this is an expression, to continue the analysis (and try to find more errors) the expression should yield a value. Here, `lvp(noloc,undefined)` is inserted as the value of the expression. Finally, the third rule represents the case where `E` evaluates to a non-null pointer that points to location `L'`. Using this location, the dereference looks up the value pointed to by the pointer using the location lookup operation, `llookup`.

Although most of the interesting logic involves expressions, to allow a more precise analysis it is useful to modify some of the statement semantics. Specifically, it is possible to eliminate some false positives by adding special logic for loops and conditionals in those situations where the programmer has already inserted null checks. An example code fragment is shown in Figure 10. Here, the assignment of `5` to `*p` should not cause a warning to be issued, since the user explicitly checks that `p` is not null, while the assignment of `10` to `*p` should still cause a warning.

To allow for this scenario, the conditional logic is overridden, with specific checks for different expression "patterns" common in null checking – the use of just the pointer, like in `if(p)`, or the use of a comparison against `NULL`, like `if(p != NULL)` or `if(NULL != p)`. In both patterns, the conditional body will only be entered when the pointer is not null, so it is possible to assume this at the start of the body. Similar logic works for loops and `if` statements with `else` branches. Note that we should not try to do the opposite, and assume a known not null pointer is null after a check like `if(!p)`, since this could lead to a false positive.

PolicyExp $PE ::=$ `null` E | `notNull` E | PE `and` PE |
PE `or` PE | PE `implies` PE | `not` PE

Figure 11: Not Null Policy Language

```
1  #include <stdlib.h>
2  //@ pre(NOTNULL): notNull(q)
3  int f(int *q) {
4    int x,*p;
5    p = (int*)malloc(sizeof(int));
6    *p = 10;
7    if (p) { *p = 20; }
8    x = *p;
9    free(p);
10   p = q;
11   x = *p;
12   return x;
13 }
```

```
line 6: Attempt to dereference a null pointer
line 8: Attempt to dereference a null pointer
```

Figure 12: Not Null Sample Run

Once the C semantics has been properly adapted for the not null policy and a not null domain has been defined, the final step is to define the policy language used in the annotations. The policy language used here is shown in Figure 11. It includes standard logical connectives (conjunction, disjunction, implication, negation) and some policy-specific keywords: `null` and `notNull`. Along with the definition of the language, rules for processing the language must be defined as well. For instance, to process `null p`, `p` must be evaluated, with the resulting symbolic value then examined to see if it represents a null pointer.

Once this has been completed, it is possible to enable the policy and check actual programs. This is done by running the program through the parser, instructing it to just emit the proper assertions and assumptions for annotations in the `NOTNULL` policy. A simple example is shown in Figure 12. Pointer `p` is allocated storage on line 5 using `malloc` and then dereferenced on line 6. Since `malloc` may return null, this results in an error message. The dereferencing on line 7 does not, since it is guarded, but line 8 again causes an error, since the dereference is no longer guarded by an `if`. On line 10, q is assigned to p; since

q has been assumed `notNull` in the precondition, the dereference of `p` on line 11 is valid and does not result in an error.

5 Case Study: Units of Measurement

A common example of a domain-specific policy, often occurring in scientific and engineering applications, is the use of units of measurement. Values or variables are assigned specific units; unit rules are then used to determine what units should be derived from various language-level operations. For instance, when multiplying two values with units U_1 and U_2 (say m and s^2), the resulting unit is the product of the two units, $U_1 U_2$ ($m\ s^2$). In many languages, including C, this information on units is *implicit*: instead of having a program-level representation, variables are assumed by the programmer to have specific units, which may (or may not) be recorded in source comments. Unfortunately, since the information is implicit, it cannot be used to ensure that unit manipulations are safe, leaving open the possibility that serious domain-specific errors will go undetected.

The possibility of serious errors is not just theoretical, as illustrated by two well-known incidents. On June 19, 1985, the space shuttle Discovery attempted to point a mirror at a spot 10,023 feet above sea level; unfortunately, the software expected the input to be in nautical miles, causing it to interpret 10,023 feet as 10,023 nautical miles, or roughly 60,900,905 feet above sea level. This caused the space shuttle to flip over mid-flight to point the mirror at the "correct" location. Roughly 15 years later, on September 30, 1999, NASA's Mars Climate Orbiter crashed into the Mars atmosphere because of a software navigation error, determined to be caused by the conflicting use of English and metric units by two different development teams [30].

In this section, we present a more complex policy than the pointer policy presented in Section 4. This policy, for units of measurement, allows unit annotations to be added to C programs, allowing the implicit information about units present in the program to be made explicit. Using these annotations, the pluggable policy includes logic to check the program to ensure that no unit violations occur, ensuring that a program which passes validation contains no unit errors.

In a fashion similar to that in Section 4, the policy is presented by showing the policy domain, policy-specific C semantics, and policy language processor. Comparisons with other techniques for checking units of measurement are deferred until Section 6.

5.1 Domain: Units of Measurement

In the International System of Units (SI), there are seven *base dimensions*, including length, mass, and time [1]. Each base dimension includes a standard *base unit*, such as meter for length or seconds for time. Other units can be defined for each dimension in terms of the base unit – feet or centimeters for

```
op _^_ : Unit Rat -> Unit .
op __ : Unit Unit -> Unit [assoc comm] .

eq U ^ 0 = noUnit .
eq U ^ 1 = U .
eq U U = U ^ 2 .
eq U (U ^ N) = U ^ (N + 1) .
eq (U ^ N) (U ^ M) = U ^ (N + M) .
eq (U U') ^ N = (U ^ N) (U' ^ N) .
eq (U ^ N) ^ M = U ^ (N * M) .
```

Figure 13: Units of Measurement

length, for instance. Units can also be combined to form derived units, such as area (meters squared, or meter meter) and velocity (meters per second).

Technically, the algebraic structure of units forms an Abelian group extended with rational powers. This is modeled in the units policy as the equational theory in Figure 13. Given named units (such as `meter`, `kilogram`, etc), new units can be formed by raising the given unit to a rational power (the first `op`) or by taking the product of the two units (the second `op`), the first being a shorthand for the repeated application of the second. The product of two units is associative and commutative, represented with the `assoc` and `comm` attributes. A number of equational rules are used to determine when two units are equal – for instance, given a unit U and that same unit to the rational power N, the product $U\ U^N$ is equal to U^{N+1}. For instance, $m\ m^2 = m^3$.

The definition in Figure 13 is also augmented with several special-purpose units not shown in the figure: `cons`, for constants; `any`, for a value that can be used with any unit; `noUnit`, for a value with no associated unit; and `fail`, representing a unit failure (like when values with different units are summed). The result of combining `fail` and any other unit is `fail`. Finally, to represent cases where the unit is unknown but generally not safe to use in combination with other units, it is possible to generate fresh units, each of which is unique. Given this definition, the values used in the units policy will be elements of the algebraic model, fresh units, and the special-purpose units, along with some auxiliary values to model pointers, structures, and arrays. The policy does not model actual values, such as numeric values assigned to integer variables.

5.2 Unit-Specific Semantics

The unit-specific semantics reflect two goals of the unit policy design: minimizing the number of required annotations, and providing a clean, intuitive semantics for units. These goals are supported through such features as the use of fresh units, to guarantee a safe unit for unannotated variables, and the association of units with values, not variables, allowing units to flow into new

variables on assignment. The latter is important for programs where variables can change units during execution, a common case in computations that use temporaries, but a case disallowed by many typed approaches, which require the type of a variable to be fixed.

Expressions Given the domain defined above, the semantics defined as part of the units of measurement policy must appropriately combine units, checking for potential errors. The general rules used by the policy verifier are:

- values used in addition and subtraction operations must have the same, or compatible, units;
- values used in multiplication, division, and mod operations may have different units, with the result having the product of the units (in the case of division and mod, the product of the inverse of the units);
- a value assigned a unit can be bitwise shifted left or right, with these operations treated identically to multiplication and division, but cannot be otherwise used in bitwise operations;
- units are associated with values, not variables, meaning assignment will change the unit of the variable (or dereferenced pointer, structure field, etc);
- most other expressions will either not change the unit (post-increment, cast, etc) or will generate a value with `noUnit` (logical or bitwise and, including with assignment).

Compatible units are generally special units which can be used with different more concrete units. For instance, in addition, if one unit is `cons`, for constants, the resulting unit will be the unit of the other operand. This allows constants to be used in any computation without a requirement to assign them a specific unit.

An example of the expression rules is shown in Figure 14. In the first, for multiplication, the resulting unit is just the product of the units of the two operands. In the second, for addition, the units need to be merged, using the unit compatibility rules (with the addition that two identical units are always compatible and a merge with `fail` always yields `fail`). If the units are not compatible, the `fail` unit will be assigned as the resulting unit, which will be detected by `checkForFail`, with an error message including the name of the operation (given in quotes) being generated. The third rule is for the greater than operation; this is similar to addition, in that incompatible units cannot be compared, with the additional step that the propagating unit is discarded (with the `discard` item) and a new unit, `noUnit`, returned. This follows the intuition that truth values (the result of greater than) do not have units. Finally, the last rule shows an assignment; here, the unit currently held in the object, `U`, is discarded, with `U'` then assigned to location `L`, the location in memory of the object being assigned into. `assignKeep` performs this assignment and also

```
eq k(val(U,U') -> * -> K)
 = k(val(u(U U')) -> K) .

eq k(val(U,U') -> + -> K)
 = k(val(u(mergeUnits(U,U'))) ->
     checkForFail("addition") -> K)

eq k(val(U,U') -> > -> K)
 = k(val(u(mergeUnits(U,U'))) ->
     checkForFail("greater than") -> discard ->
     val(u(noUnit)) -> K) .

eq k(val(lu(L,U),U') -> = -> K)
 = k(val(U') -> assignKeep(L) -> K) .
```

Figure 14: Units Expression Rules, in Maude

keeps the resulting unit in the computation, allowing it to propagate for cases like `x = y = z`.

Statements and Declarations Most of the statement-processing rules from the framework semantics are reused directly. This includes the rules for handling features such as conditionals, loops, switch statements, and gotos, which can cause environment splitting. The rules for declarations are unit-specific, though, so they need to be extended as part of the unit policy. When CIL simplifies the C source, it moves all declarations to the top of the function, including those nested in blocks, using renaming to enforce proper scoping. It also moves initializations so they occur after declarations – i.e., declarations like `int x = 5;` are transformed into `int x; x = 5;`. When the declaration occurs, a default fresh unit is assigned to each declarator of scalar (non-pointer) type, or each field for structure types; pointers are assigned a location which, when dereferenced, holds a fresh unit. This use of fresh units allows undefined variables to be combined in safe ways (such as in products), but prevents dangerous combinations (such as sums). On assignment, the fresh unit value will be overwritten by the value of the right-hand side expression.

5.3 Unit Policy Language

The unit policy language is shown in Figure 15. The logical portion of the language, represented by `UnitExp`, is similar to that described for the pointer policy language in Section 4, with the same logical connectives. In addition, a `unit` operation over expressions, `unit`, is defined; this operation represents the current unit of expression E. Since units are a rather complex value domain, it would be impossible to have an operator associated with each possible unit value (like `null` and `notNull` from the prior policy), so the policy language

$$
\begin{array}{lrl}
\textit{Unit} & U ::= & \texttt{unit(}E\texttt{)} \mid \texttt{unit(}E\texttt{)} \wedge\ Q \mid BU \mid U\ =\ U \mid U\ U \\
\textit{UnitExp} & UE ::= & U \mid UE\ \texttt{and}\ UE \mid UE\ \texttt{or}\ UE \mid \\
 & & UE\ \texttt{implies}\ UE \mid \texttt{not}\ UE
\end{array}
$$

Figure 15: Units Policy Language

also defines an equals operator, `=`. In preconditions and assumptions, this is treated like assignment – assuming a variable has a certain unit is treated like an assignment of that unit value to the variable. In postconditions and assertions, this is treated like an equality check, where the current unit of expression E in `unit(E)` is compared to the unit on the other side of the equality. In many situations it is useful to include a rational exponent on a unit expression, so this is allowed as well. One example of where this is useful is for stating formulas like `unit(x)^2 = unit(y) unit(z)`, which is a more concise and easier to read way of stating `unit(x) = (unit(y) unit(z))^1/2`. Base units, `BU`, are also included as units, allowing the use of defined units and unit combinations such as `m` for meter or `m s^2` for meter second squared. Finally, a new unit can be formed from existing units, such as was seen in the exponent example, with `unit(y) unit(z)` representing the product of the units of `y` and `z`.

5.4 Unit Verification

Unit verification takes place using the verification process described in Section 3. Here two other issues are addressed: correctness of the analysis and performance.

Correctness The semantic rules in the units policy have been chosen to be conservative and, in some cases, fairly restrictive, to ensure that any unit safety violations are discovered. For instance, the policy semantics does not allow unit changes through pointers without an explicit assumption. The policy also assumes a type-safe subset of C, since violations of type safety can easily invalidate the analysis results. Outside of this, the policy encoding is a fairly direct encoding into rewriting logic, with the encoding of the units of measurement group shown in Figure 13 and with one policy-specific rule added for each C expression. An example of this was seen in Figure 14, with rules for multiplication, addition, greater than, and assignment expressions. It is possible, with the given encoding and restrictions, to state the following:

Theorem 1. *Given a program P written in a type-safe subset of C, any unit safety violations in P are reported; if no violations are reported, program P is unit-safe.*

This merits several comments. First, as mentioned above, this does not necessarily hold for code which is not type-safe, since it would be possible to alter

			Total Time		Average Per Function		Average Per Statement	
Test	LOC	Stmts	x100	x4000	x100	x4000	x100	x4000
straight	25	68	6.19	226.65	0.06	0.06	0.0009	0.0008
ann	27	80	7.6	287.66	0.08	0.07	0.0010	0.0009
nosplit	69	152	13.13	511.24	0.13	0.13	0.0009	0.0009
split	69	264	49.62	1475.44	0.50	0.37	0.0019	0.0014

Single 3.40 GHz Pentium 4, 2 GB RAM, OpenSuSE 10.2, kernel 2.6.18.8-0.7-default, Maude 2.3. Times in seconds. All times averaged over three runs of each test. LOC (lines of code) and Stmts (statements) are per function, with 100 or 4000 identical functions in a source file.

Figure 16: Unit Safety Verification Times

values in ways that would invalidate the analysis, such as with pointer arithmetic or unions. Second, this blindly assumes programmers' assumptions. Of course, if programmers do add false or unintended assumptions, the verification can determine that a program with unit safety violations is actually unit correct. Third, while this theorem states that all violations are discovered, it does not state that false positives do not occur; it is still possible for programs to report violations that could not actually occur at runtime. This is partially a limitation of the conservative assumptions made because of issues like aliasing, and partially due to the symbolic nature of the execution – branches which are dead will still be taken, for instance, and can still lead to violations. Finally, correctness theorems do depend on the policy, but actually depend little on the core; since values are defined at the policy-level, the framework leaves rules for combining values and determining the results of expressions to the policies themselves.

Performance While correctness is critical, it is also important that the unit policy is able to check program correctness in a reasonable amount of time. Figure 16 shows performance figures for the unit safety policy. Here, each test performs a series of numerical calculations: `straight` includes straight-line code; `ann` includes the same code as `straight` with a number of added annotations processed using the policy language; `nosplit` includes a number of nested conditionals that change units on variables uniformly, leaving just one environment; and `split` includes nested conditionals that change variable units non-uniformly in different branches, yielding eight environments. LOC gives the lines of code count, derived using the CCCC tool [23], for each function, with the same function repeated 100 or 4000 times. Stmts instead gives the number of statements in the function after the code is transformed into three-address form; this is a better indicator of the volume of work, since each statement must be individually checked.

Several performance figures stand out. First, performance scales almost linearly. Second, performance scales well on a per-statement basis. This can be seen with `straight`, `ann` and `nosplit`, for instance, where all three take roughly the same per-statement time during verification. Third, splitting environments increases the per-statement execution time, but not prohibitively. With eight

environments, the time per statement to process split is roughly double that to process nosplit, which has just one environment, versus taking eight times longer, a potential worst case with eight times as many environments. Fourth, processing annotations, even using a custom annotation language, seems to add little overhead; annotations are treated as statements during processing, but do not increase the per-statement processing time average.

For statements within a single function, performance also scales well. With a single instance of the function from `straight`, processing time is 0.59 seconds, or 0.0087 seconds per statement. Duplicating the calculations in this function to extend it from 25 LOC to 3025 LOC, with 9068 statements instead of 68, overall processing time is 28.35 seconds, with only 0.0031 seconds per statements. This indicates an economy of scale, with startup costs amortized over an increasing number of statements, and indicates that unit policy checking can be performed on even large functions in a realistic amount of time.

6 Related Work

Numerous approaches have been proposed for static analysis, including pointer analysis, in the context of C and other languages [32, 16, 8, 35]. We argue that, as illustrated in this paper, many of these approaches can be captured as pluggable policies in our framework, provided that proper symbolic rules are specified. The simple null pointer checker shown in this paper is inspired by work on the LCLint tool[11], but does not (yet) include the same level of functionality[2]. We leave more advanced pointer analyses as part of our future work.

Units of Measurement Related work on unit safety tends to fall into one of three categories: library-based solutions, where libraries which manipulate units are added to a language; annotation-based solutions, where user-provided annotations assist in unit checking; and language, including type system, extensions, where new language syntax or typing rules are added to support unit checking in a type-checking context.

Library-based solutions have been proposed for several languages, including Ada [17, 24], Eiffel [20], and C++ [7]. One significant library for unit safety was developed by the Mission Data Systems team as NASA's JPL. This library, written in C++, included several hundred classes representing typical units, like `MeterSecond`, with appropriately typed methods for arithmetic operations. An obvious disadvantage of this approach is that, to include new units, new classes must be added, and existing classes must be extended with new methods to properly type arithmetic expressions with the new unit.

Systems based on annotations include the system that served as an inspiration for this work, C-UNITS [31]. The C-UNITS system also provides for unit annotations on function headers and in function bodies, like the unit safety

[2]Some additional functionality, to deal with undefined pointers and deallocated memory, is already present in an extended version of the checker presented here.

policy presented here. However, C-UNITS is not scalable; units are determined by symbolically executing the entire program, instead of the per-function approach taken here, making analysis of large code bases expensive and, in some cases, impossible. Also, C-UNITS is a unit specific solution, while the framework presented here supports units as one of many possible policies.

Solutions based around language and type-system extensions work by introducing units into the type system and potentially into the language syntax, allowing expressions to be checked for unit correctness by a compiler or interpreter using extended type-checking algorithms. MetaGen [3], an extension of the MixGen [2] extension of Java, provides languages features which allow the specification of dimension and unit information for object-oriented programs. Other approaches making use of language and type system extensions have targeted ML [22, 21], Pascal [13, 18], and Ada [14]. These solutions differ from that presented here in that the framework and units policy make no changes to the underlying language or type system, with all checking driven by the annotations.

A newer tool, Osprey [19], also uses a typed approach to checking unit safety, allowing type annotations in C programs (such as `$meter int`) which can then be checked using a combination of several tools, including the annotation processor, a constraint solver, a union/find engine, and a Gaussian elimination engine (the latter two used to reduce the number of different constraints and properly handle the Osprey representation of unit types). The approach taken by Osprey is quite different from the approach taken in the unit policy presented here. First, the use of unit annotations in types requires a change to the C language, something that we have avoided. Second, Osprey annotations do not provide a language as rich as that provided by our policy. For instance, in the unit policy, it is possible to declare that the unit of a function result is related to that of the input parameter. An example is with a square root function, where the unit of the result squared is the unit of the input parameter:

```
//@ post(UNITS): unit(@result)^2 = unit(x)
double sqrt(double x) { ... }
```

It is not possible to express such constraints in Osprey without manually editing files generated during the checking process. Third, Osprey, and other type-based approaches, do not allow changing the units of a variable during a computation, something which can occur often when temporaries are reused in calculations. Fourth, Osprey checks at the level of *dimensions*, not units, treating units such as feet or pounds as meters or kilograms with conversion factors. We believe this masks some potential errors, such as when a user thinks a function uses feet while it actually uses meters. Even if conversions are inserted (which Osprey does not seem to do), this still masks a misunderstanding about the called function. Finally, we believe the error reporting capabilities of the framework, and with it the unit policy, provide more accurate feedback to the programmer, something that can be especially important in large code bases.

7 Future Work and Conclusions

In this paper we presented a framework for building pluggable policies for domain-specific verification and analysis of C programs. We have illustrated this framework by showing two such policies. The first, a not null checker, provides an annotation language and analysis to check programs for improper pointer dereferences. The second, a unit of measurement checker, provides a richer annotation language than the first, allowing unit properties of C programs to be declared and verified. A large amount of related work exists, both in terms of these specific analyses and program verification in general; we believe comparisons of this work with ours show that we can perform similar analysis with a more modular framework, allowing multiple, quite different analyses to be written using the same tools and run over the same programs.

In the future, we would like to look at other potential policies that would fit well within this system. Of special interest is analysis of coordinate systems, another problem domain taken from scientific computing. We would also like to analyze the connections between this technique and that of pluggable types [6], since we believe it would be possible to represent pluggable types as annotations in our current system with a fairly straight-forward translation. Finally, we would like to improve the existing analysis capabilities, extending the framework to handle some common scenarios outside of a type-safe core of C, improving alias information, and increasing performance.

References

[1] The NIST Reference on Constants, Units, and Uncertainty. `http://physics.nist.gov/cuu/Units/`.

[2] E. Allen, J. Bannet, and R. Cartwright. A First-Class Approach to Genericity. In *OOPSLA'03*, pages 96–114, New York, NY, USA, 2003. ACM Press.

[3] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and J. Guy L. Steele. Object-Oriented Units of Measurement. In *OOPSLA'04*, pages 384–403, New York, NY, USA, 2004. ACM Press.

[4] Anonymous. Citation omitted to maintain anonymity. Please contact the PC chair for the URL of the download site.

[5] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.

[6] G. Bracha. Pluggable type systems. Revival of Dynamic Languages workshop at OOPSLA 2004, October 2004.

[7] W. E. Brown. Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation, 2001.

[8] A. C. Chou. *Static analysis for bug finding in systems software*. PhD thesis, 2003.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.

[11] D. Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, 1996.

[12] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, 1999.

[13] N. H. Gehani. Units of Measure as a Data Attribute. *Computer Languages*, 2(3):93–111, 1977.

[14] N. H. Gehani. Ada's Derived Types and Units of Measure. *Software Practice and Experience*, 15(6):555–569, 1985.

[15] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[16] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 168–181. ACM, 2003.

[17] P. N. Hilfinger. An Ada Package for Dimensional Analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, 1988.

[18] R. T. House. A Proposal for an Extended Form of Type Checking of Expressions. *The Computer Journal*, 26(4):366–374, 1983.

[19] L. Jiang and Z. Su. Osprey: a practical type system for validating dimensional unit correctness of c programs. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 262–271. ACM, 2006.

[20] M. Keller. EiffelUnits, 2002. `http://se.inf.ethz.ch/projects/markus\_keller/EiffelUnits.html`.

[21] A. J. Kennedy. Relational parametricity and units of measure. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*. Association of Computing Machinery, Inc, January 1997. Paris, France.

[22] A. J. Kennedy. *Programming Languages and Dimensions*. PhD thesis, St. Catherine's College, University of Cambridge, November 1995.

[23] T. Littlefair. C and c++ code counter. `http://sourceforge.net/projects/cccc`.

[24] G. W. Macpherson. A reusable ada package for scientific dimensional integrity. *ACM Ada Letters*, XVI(3):56–69, 1996.

[25] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[26] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[27] J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, volume 3097 of *LNAI*, pages 1–44. Springer, 2004.

[28] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[29] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[30] M. C. Orbiter. `http://mars.jpl.nasa.gov/msp98/orbiter`.

[31] G. Roşu and F. Chen. Certifying measurement unit safety policy. In *Proceedings, International Conference on Automated Software Engineering (ASE'03)*, pages 304 – 309. IEEE press, 2003.

[32] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 12–23. ACM, 2001.

[33] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

[34] E. Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.

[35] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. *SIGSOFT Softw. Eng. Notes*, 30(5):115–125, 2005.