A Rewriting Logic Approach to Static Checking of Units of Measurement in C [1]

Mark Hills[2] Feng Chen[3] Grigore Roşu[4]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, US

Abstract

Many C programs assume the use of implicit domain-specific information. A common example is units of measurement, where values can have both a standard C type and an associated unit. However, since there is no way in the C language to represent this additional information, violations of domain-specific policies, such as unit safety violations, can be difficult to detect. In this paper we present a static analysis, based on the use of an abstract C semantics defined using rewriting logic, for the detection of unit violations in C programs. In contrast to typed approaches, the analysis makes use of annotations present in C comments on function headers and in function bodies, leaving the C language unchanged. Initial evaluation results show that performance scales well, and that errors can be detected without imposing a heavy annotation burden.

Keywords: Unit safety, rewriting logic, abstract semantics, static analysis.

1 Introduction

Many programs make use of domain-specific information. A common example, often occurring in scientific and engineering applications, is the use of units of measurement. Units are associated with specific values or variables; unit rules then determine how operations in the language (addition, multiplication, etc) change and combine units, and also determine when this is safe. In many languages, including C, this information on units is *implicit*: instead of having a program-level representation, values are assumed by the programmer to have specific units, which may be documented informally in source comments. Unfortunately, the implicit nature of this information means that it cannot be used to automatically ensure that unit manipulations are safe, i.e., that operators are always applied to operands with

[1] Supported by NSF CCF-0448501 and NSF CNS-0509321.
[2] Email: mhills@cs.uiuc.edu
[3] Email: fengchen@cs.uiuc.edu
[4] Email: grosu@cs.uiuc.edu

```
typedef struct {
  double atomicWeight;
  double atomicNumber;
} Element;

//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ post(UNITS): unit(@result) = m ^ 2 kg ^ -1
double radiationLength(Element * material) {
  double A = material->atomicWeight;
  double Z = material->atomicNumber;
  double L = log( 184.15 / pow(Z, 1.0/3.0) );
  double Lp = log( 1194.0 / pow(Z, 2.0/3.0) );
  return ( 4.0 * alpha * re * re) * ( NA / A ) *
         ( Z * Z * L + Z * Lp );
}

//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ pre(UNITS): unit(density) = kg m ^ -3
//@ pre(UNITS): unit(thick) = m
//@ pre(UNITS): unit(initEnergy) = kg m ^ 2 s ^ -2
double finalEnergy(Element * material, double density,
                   double thick, double initEnergy) {
  double X0 = radiationLength(material);
  return initEnergy / exp ( thick / X0 );
}
```

Fig. 1. Electron Energy Example, in C

compatible units. The burden to ensure this falls directly on the programmer. This leaves open the possibility that serious domain-specific errors will go undetected.

The possibility of serious errors is not just theoretical. On September 30, 1999, NASA's Mars Climate Orbiter spacecraft crashed into Mars' atmosphere due to a software navigation error, caused by one team using English units while another used metric units in a key Orbiter operation [2]. Roughly 15 years before this, the space shuttle Discovery flipped over mid-flight in an attempt to point a mirror at a spot 10,023 feet above sea level; the software interpreted this figure as 10,023 nautical miles, or roughly 60,900,905 feet [27].

Checking by hand may be an option for small programs, but does not scale to large programs. Even in small programs, some calculations can be very complex and can depend on non-local information, like the contents of global variables and the results of function calls, making manual checking challenging. For instance, a portion of a program used to calculate the final energy of an electron [5] is shown in Figure 1. Without a method to record expected units and check for correctness,

[5] This example was borrowed from Jiang and Su's work on Osprey [18], which in turn borrowed it from Brown's work on SIUNITS [7].

it is not obvious whether the code is, or is not, unit-safe. In fact, line 26 will report a unit error: the unit returned by the `radiationLength` calculation will be $\texttt{meter}^2\texttt{kilogram}^{-1}$, and `thick` has unit `meter`, so `thick` divided by `X0` will have unit $\texttt{meter}^{-1}$ `kilogram`. However, `exp` expects a unitless argument, meaning either the annotations are incorrect or `radiationLength` is not being used correctly.

Many approaches have been proposed to enforce unit safety in programs, a number of which are discussed in Section 2. In this paper, we propose a new solution for the C language, CPF[UNITS]. CPF[UNITS], based on the CPF framework [16], is a significant extension of the ideas introduced in the proof-of-concept C-UNITS system [29]. CPF[UNITS] allows unit-specific annotations to be added to C programs in the form of function preconditions, function postconditions, assertions, and assumptions. These annotations are then checked for validity using a combination of the abstract rewriting logic semantics of C, part of CPF, and the UNITS *policy*, a collection of unit-specific semantics for certain language features and the combination of an annotation language and annotation semantics. Hence the name CPF[UNITS], for CPF parameterized by the UNITS policy.

The remainder of this paper is organized as follows. We first present related work in Section 2, including the earlier C-UNITS system and approaches based on types. We then provide introductory details on the abstract rewriting logic semantics of C in Section 3, assuming familiarity with term rewriting and a basic familiarity with equational or rewriting logic. An introduction to units of measurement in presented in Section 4, followed by details of the CPF[UNITS] unit safety checker in Section 5. Section 6 presents initial evaluation results, with Section 7 presenting possible future work and concluding. Our website provides downloads of all tools and examples described in this paper, along with a web-based interface for experimentation [1].

2 Related Work

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; language and type system extensions, where new language syntax or typing rules are added to support unit checking in a type checking context; and annotation-based solutions, where user-provided annotations assist in unit checking.

Library-based solutions have been proposed for several languages, including Ada [15,23], Eiffel [19], and C++ [7]. The Mission Data Systems team at NASA's JPL developed a significant library, written in C++, which includes several hundred classes representing typical units, like `MeterSecond`, with appropriately typed methods for arithmetic operations. An obvious disadvantage of such an explicit approach is that the units supported by the library are fixed: adding new units requires extending the library with new classes and potentially adding or modifying existing methods to ensure the new classes are properly supported.

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 [5], an extension of the MixGen [4] extension of Java, provides language 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 [21,20], Pascal [13,17], and Ada [14].

A newer tool, Osprey [18], also uses a typed approach to checking unit safety, allowing type annotations in C programs (such as `$meter int`) using a modified version of CQUAL [12]. These annotations 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 as matrices). One limitation of Osprey is that there is no way to express relationships between the units of function parameters and return values, something possible with a richer annotation language:

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

Instead, this type of relationship has to be added by hand-editing files generated during processing. Osprey also checks *dimensions* (i.e., length), not units (i.e., meters or feet), instead converting all units in a single dimension into a *canonical* unit. This can mask potential errors: for instance, it is not an error to pass a variable declared with unit meter to a function expecting feet. On the other hand, Osprey includes functionality to check explicit conversions for correctness, catching common conversion errors such as using the wrong conversion factor.

Annotation-based systems, including JML [8] and Spec# [6], have been applied to many problem domains, but not specifically to units. Systems for unit safety based on annotations include the C-UNITS system [29], which used concepts about abstract semantics and annotations that first appeared in the context of BC, a small calculator language [9]. CPF[UNITS] was inspired by the work on C-UNITS, and takes a similar approach, with a focus on using abstract semantics and annotations. However, CPF[UNITS] has extended this approach in three significant ways. First, CPF[UNITS] has been designed to be *modular*: the abstract semantics of C have been completely redefined using concepts developed over the last several years as part of the rewriting logic semantics project [25]. The semantics are divided into *core* modules, shared by all CPF policies, and *units* modules, specific to CPF[UNITS]. This allows improvements in the core modules to be shared by all policies, simplifies the unit checking logic, and greatly improves the ease with which the semantics can be understood and modified. Second, CPF[UNITS] has been designed to cover a much larger portion of C. C-UNITS was designed as a prototype, and left out a number of important C features, with minimal or no support for structures, pointers, casts, switch/case statements, gotos, or recursive function calls. Support for expressions was also limited, with the main focus on commonly-used expressions, and more complex memory scenarios (structures with pointers, arrays of pointers, etc) were ignored. CPF[UNITS] supports all these features, and makes use of a more advanced parser to handle a larger class of C programs. Finally, CPF[UNITS] has been designed to be more scalable. While C-UNITS requires a complete program for analysis, CPF[UNITS] analyzes individual functions, leading to smaller individual verification tasks.

$$\mathtt{k}(\underset{\mathtt{lvp(L,U)}}{\underline{\mathtt{lookup(X)}}}\rangle\ \mathtt{env}\langle[\mathtt{X,L,U}]\rangle$$

Fig. 2. Sample C Semantic Rule, in K

```
eq k(lookup(X)          -> K) env(Env [X,L,u(U)])
 = k(val(lvp(L,u(U))) -> K) env(Env [X,L,u(U)]) .
```

Fig. 3. Sample C Semantic Rule, in Maude

The technique used by CPF[UNITS], like most (if not all) static analyses, could be framed in terms of abstract interpretation [11], where the domain of interpretation is the algebra of units of measurement. However, CPF[UNITS] makes intensive use of recently developed rewriting logic language definitional techniques based on representations of abstract syntax trees as continuations; establishing the relationships between rewriting logic semantics and abstract interpretation is an interesting subject in and of itself, but it goes beyond our purposes in this paper.

3 Abstract Rewriting Logic Semantics of C

The abstract semantics of C is defined using Maude [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 abstract values and other information), 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 at the right. This methodology is described in more detail in papers about the rewriting logic semantics project [24,25]. To simplify the presentation of the rules in the CPF semantics, we use K notation [28], which includes a number of simplifying conventions.

Figures 2 and 3 show an example of a semantic rule included in the abstract C semantics used in the CPF[UNITS] tool (see Section 5), first in K notation, then in Maude. The rule represents a memory lookup operation. Here, if identifier `X` is being looked up, and the environment contains an item named `X` with location `L` and unit value `U`, a location value pair `lvp` containing `L` and `U`, `lvp(L,U)`, is returned in place of the lookup operation, while the environment remains unchanged [6]. The K version uses three K conventions: `>` is used in place of `)` to represent "and everything else", expanded into `-> K` in the Maude version; `<_>` is used for set matching ("everything else to either side"), which requires `Env` to represent "everything else" in Maude; and replacement is represented by underlining the portion of the term that has changed, allowing unchanged portions of the term to be mentioned without the need to be repeated. K also does not need the wrapper `u()`, which is used to wrap units and turn them into values.

[6] This is slightly simplified from the actual semantics, where the environment is made up of 5-tuples instead of triples, but is otherwise the same.

```
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 ^ Q) = U ^ (Q + 1) .
eq (U ^ Q) (U ^ P) = U ^ (Q + P) .
eq (U U') ^ Q = (U ^ Q) (U' ^ Q) .
eq (U ^ Q) ^ P = U ^ (Q * P) .
ops noUnit any fail cons : -> Unit .
ops meter m feet f : -> Unit .
```

Fig. 4. Units of Measurement, in Maude

4 Units of Measurement

In the International System of Units (SI), there are seven *base dimensions*, including length, mass, and time [3]. Each base dimension includes a standard *base unit*, such as meters for length or seconds for time. Other units can be defined for each dimension in terms of the base unit – feet or centimeters for 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. This provides several important properties which need to be modeled during unit checking. First, as mentioned above, units can be combined to form new units – for any two units A and B, AB is also a unit (AB is the product of units A and B). Units are also associative (given C is also a unit, $(AB)C$ is the same as $A(BC)$), commutative (AB is the same as BA), and have inverses and identities. Generally, products of the same unit are represented with exponents, i.e. AA is the same as the more commonly used A^2, but both forms are acceptable and should be usable. Our equational definition of the units domain is shown in Figure 4. The first two operands (defined with `op`) specify that a unit can have a rational exponent and that the product of two units is a new unit. The following seven equations (defined with `eq`) are used to simplify units, putting them into a canonical form, with `P` and `Q` representing rational numbers. The next two operand lists define some actual units: `meters` and `feet`, along with short forms, plus special units: `noUnit`, `any`, `fail`, and `cons`. `noUnit` represents the unit of values that have none, like the result of a bitwise computation. `any` means a value can be considered to be of any unit, which is similar to `cons`, the unit given to constants (`cons` is used internally as the default unit of constants, while `any` can be used in annotations). Finally, `fail` represents a unit failure, and is represented as a unit so it can be easily propagated. Additional equations, not shown here, are also provided to allow for canonical forms of predefined units, for instance ensuring that `m` and `meter` are recognized as the same unit.

5 CPF[UNITS]: Checking Unit Safety of C Programs

In this section we present CPF[UNITS], a tool for checking the unit safety of C programs. In CPF[UNITS], users specify units on C *objects*[7] that hold numerical values by providing annotations in comments in the source code. Annotations indicate function preconditions and postconditions, assertions, and assumptions. The annotated code is converted into a formal representation based on a Maude-defined abstract C semantics, and then checked function by function, ensuring that the size of the verification task does not (except in some pathological conditions, such as with deeply nested conditionals where each branch makes different changes to units) grow too large. The use of conditionals and looping constructs can cause multiple units to be associated with a single object [8]. Techniques to handle this, while still maintaining precision, are discussed below.

CPF includes logic to add annotations to C programs, parse these programs, generate verification tasks, and process most C statements. The portions of CPF not specific to the UNITS policy are described below at a high level; additional coverage, including detailed information about the CPF, a sample not-null policy, and a high-level introduction to the units policy can be found in a companion technical report [16].

5.1 Code Annotations

In CPF[UNITS], code annotations are included directly in the C source code as comments, starting either with `/*@` (for block comments) or `//@` (for line comments). Examples of annotations can be found in Figure 1. Note that both functions, such as `radiationLength` and `finalEnergy`, and function prototypes can be annotated, allowing annotations to easily be added to library functions. `@result` is a special token used to refer to the return value of the function. In general, a function can have multiple, or no, preconditions or postconditions.

5.2 Generating Verification Tasks

The CPF[UNITS] semantics assume that one verification task will be generated for each function. The tasks are generated using a combination of a Perl script for processing annotations and a modified version of the CIL tool for C [26], which provides parsing, analysis, and code transformation capabilities. After parsing the program, CIL first performs a CPF-specific three-address transform, moving expression computations out of function calls and return statements. CIL then performs a CPF-specific inlining step, where function call sites are replaced by the function preconditions and postconditions of the called function, with preconditions becoming asserts and postconditions becoming assumes. The preconditions and postconditions for each function are then moved into the function body, with preconditions becoming assumptions at the start of the body and postconditions placed before each return statement as assertions. The latter operation is safe because the trans-

[7] In C, an object is a memory region that can be read from or written to.

[8] Recursive calls do not require similar treatment; since functions are checked individually, call sites are handled without descending into the called function.

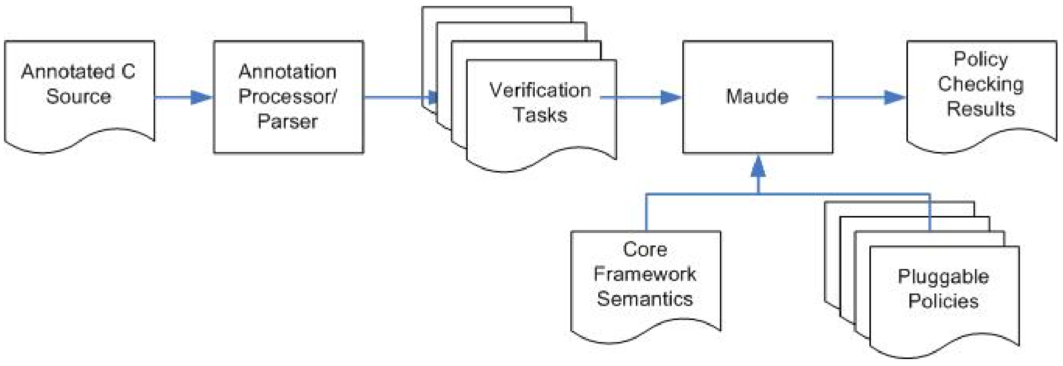

Fig. 5. CPF Framework Execution Model

form moves any computation out of the return statement, ensuring that a return will not also modify units referenced in the postcondition. Finally, CIL also generates the verification tasks as part of a CPF-specific pretty-printing step; instead of printing out modified C code, which is the standard CIL behavior, verification tasks for each function, given in Maude using the CPF abstract C syntax, are generated. Figure 5 illustrates the process of checking annotated code using CPF. More information about this process can be found in the CPF technical report [16].

5.3 Checking Unit Safety

Once the verification tasks for each function have been generated, each task is checked using Maude. The executable nature of rewriting logic specifications is leveraged to symbolically execute programs using an abstract rewriting logic semantics. This semantics is made up of the CPF core semantics and, for unit safety, the unit-specific semantics included in the UNITS policy. The CPF semantics includes an abstract syntax of C and semantics for C statements, high-level definitions of concepts such as "value" and "annotation language", and a number of *hooks* which allow policy-specific behavior to be added. CPF[UNITS] extends this with semantics for declarations, assertions, assumptions, and expressions, a definition of the UNITS annotation language used in annotations, and a policy-specific concept of unit values. Here, we focus on those parts specific to CPF[UNITS], with the CPF semantics described here at a high level of detail; as mentioned above, more detail about CPF is available in a technical report [16] and on the CPF website [1].

5.3.1 CPF: Shared Semantics

The bulk of the shared CPF semantics is focused on the semantics of C statements, defined over an abstract semantics for C language constructs. In CPF, statements are executed in an *environment*, which provides information on the names, values, and types of C objects which will be used in a statement's constituent expressions. In some cases, such as with conditionals that make different changes to the environment on different paths, it is possible for a statement to start with one environment but return multiple environments as the result of execution[9]. Part of the program state is thus a *set* of environments, with the framework executing each statement

[9] This makes our analysis *path-sensitive*, although we make no attempt to track which conditions led to which path.

```
1  int x,y,z;
2  //@ assume(UNITS): unit(x) = unit(y) = m
3  if (b) {
4    y = 3; //@ assume(UNITS): unit(y) = f
5    x = y;
6  }
7  z = x + y;
```

Fig. 6. Path-Sensitive Unit Assignment, in C

```
op _^_ : Unit CInt -> Unit .
op u : Unit -> Value .
op ptr : Location -> Value .
op arr : Location -> Value .
op struct : Identifier SFieldSet -> Value .
op union : Identifier SFieldSet -> Value .
```

Fig. 7. CPF[UNITS] Values, in Maude

once in each environment, and gathering the resulting environments together as the environment set to use for the following statement(s). An example where this could occur is shown in Figure 6, where the unit assigned to both `x` and `y` starts off as `m`, but can be either `m` or `f` for both at the end of the `if`. If, instead of using environment sets, sets of values were associated with each object, a false positive would be generated on line 7, since `x` and `y` could both either be `f` or `m`, leading to an invalid combination of one of each. One disadvantage to this use of environment sets is that the analysis can be potentially more expensive, especially in certain pathological cases where the environment keeps splitting (deeply nested conditionals, for instance). In practice this does not appear to happen often, since units do not often change once they are initially assigned; as a precaution, CPF allows the use of a high-water mark on the size of the environment set, which when crossed causes some environments (selected randomly) in the set to be discarded and a warning message to be issued.

CPF defines semantics for all C statements, such as conditionals and gotos, and also includes semantics to "break down" expressions into their constituent pieces: `E + E'` into evaluations of `E` and `E'`, for instance. The policy-specific semantics specifies the abstract values to which `E` and `E'` can evaluate.

5.3.2 *CPF[UNITS]: Abstract Values*

The CPF[UNITS] values are a combination of unit values and values representing C pointers, structures, unions, arrays, and enums (treated as constants). The unit values are those defined in the theory of units shown in Figure 4, augmented with C-specific unit values that include simple C expressions, such as constant integers in the exponent. Pointers are represented as locations; dereferencing accesses the actual value at that location, such as a unit, a structure, or another pointer (for multiple levels of indirection). Arrays have a similar representation, which allows them to be used like pointers, but limits them to containing only a single value, so all array elements are considered to have the same unit. Structures and unions contain the name of the structure or union type (anonymous structures and unions are given names by CIL) and a set of field/location pairs to indicate where the value of each field is stored. Function pointers are represented with a special value, with alias analysis used to determine which function is invoked by an indirect call through the pointer; warnings are issued if it is not possible to determine a unique function at a call site. A subset of the value definitions used by the CPF[UNITS] policy is shown in Figure 7.

5.3.3 CPF[UNITS]: C Declarations

The CPF[UNITS] rules for handling declarations provide an initial symbolic representation of memory for the global variables, formal parameters, and local variables of a function. Different allocators are used for each C object type, allocating initial values appropriate to the object. For instance, the allocator for scalars initially associates a "fresh" unit, unique to that declaration, with the scalar, while the allocator for pointers associates a reference to a new memory location. Declarations for structure and union variables allocate field/location maps based on the fields contained in the structure or union declaration. Allocation is recursive; structure allocation also allocates the fields of the structure, while allocation of arrays and pointers allocates the base type of the array or pointer as well, with unions currently represented like structures (i.e., we do not attempt to just allocate one location shared by all fields in the union). One challenging but common case to represent is structures which contain pointers to other structures. It is not possible to allocate the entire memory representation up front, since this could be (in theory, at least) infinite. Pointers inside structures are instead created with an allocation trigger, which will allocate the pointer's target on the first attempt to access it. This allows the memory representation to grow sensibly, modeling just what is needed to perform unit checking.

After processing all declarations in the function body/verification task (CIL moves all declarations to the top of a function, using renaming to model shadowing), initial values are given to local variables using a combination of assertions (from annotations) and assignments. For instance, an assertion may indicate that variable `x` has unit `meter`; a declaration like `int y = x;` would then associate `meter` with `y` as well. Initial units for formal parameters and global variables are based just on the function preconditions. If a precondition or assignment does not indicate the initial unit of a variable, it keeps its assigned fresh (unique) unit, which will allow detection of errors from misuse of the variable in expressions. After all initial assignments are complete, a locking process locks certain memory locations to make sure they cannot be changed in ways that are not reflected in the annotations. For instance, it is not possible to write a new unit through a pointer given as a formal parameter. This ensures that changes visible outside the function but not included in the preconditions and postconditions are prevented, allowing checking to be truly modular[10]. Finally, a "checkpoint" is taken, saving the original assigned values

[10] It is possible to override this locking behavior using annotations, but this will generate a warning message to alert the user to the potential unsoundness created by doing so. We are working on incorporating alias analysis results into the locking process to ease this restriction.

```
[1] U * U' = U U'

[2] U + U' = mergeUnits(U,U') -> checkForFail("+")

[3] U > U' = mergeUnits(U,U') -> checkForFail(">") -> discard -> noUnit

[4] (lvp(L,V) = V') = V' -> assign(L)

[5] (lvp(L,U) += U') = mergeUnits(U,U') -> checkForFail("=") -> assign(L)

[6] *(lvp(L,ptr(L'))) = llookup(L')

[7] lvp(L,struct(X', (sfield(X,L') _))) . X = llookup(L')
```

Fig. 8. CPF[UNITS] Expression Rules, in K

before any changes are made in the function body. This allows these original values to be accessed later during unit checking, such as when checking the assertions added to represent the function postconditions.

5.3.4 CPF[UNITS]: C Expressions

To evaluate expressions in CPF[UNITS] the semantic rules need to properly modify, combine, and propagate abstract values representing units and C objects (pointers, structures, etc). Expressions, along with assert statements, are also the point where unit safety violations are discovered, so semantics for expressions which can produce failures need to ensure that the failures are properly handled. Figure 8 includes rules for a representative set of expressions, illustrating how abstract values are propagated and failures are detected.

The first rule models the multiplication operation. Here, given two unit values `U` and `U'`, the result is their product, `U U'`. The second rule, for addition, is structured similarly to the first, but must also check that the combination of the units is correct. This is done by merging the units with `mergeUnits`. In merging, if one unit is `any` or `cons`, both of which can be treated as being of any unit, the other unit is returned. Otherwise, the two units must match, with no automatic conversions between units performed. If the units do not match, or one of the units is `fail`, `fail` is returned to indicate a unit safety violation. This enforces the unit rule for addition – the units of both operands must be the same. To detect the failure and issue a warning, `checkForFail` is used, which will print an error message if the result unit is `fail`.

The third rule handles the greater-than relational operation. Here, the rule is the same as for addition: to compare two values they must have the same unit. Beyond this, the returned unit is `noUnit`, since it does not make sense to assign a unit to the result of the comparison. The fourth rule is used for assignment. Here, the lvalue evaluates to an `lvp`, or *location-value pair*, with the location and current value of the object being assigned into. The value of the right-hand expression is assigned over the current value to the same location. While this works for units, it also works for other C entities, such as the representations for pointers and structures. The

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

Fig. 9. Units Annotation Language

fifth rule, for the += operand, is a combination of the rules for + and assignment, performing both the check for failure and the assignment to the location of the lvalue. In this case, the values should be units, since it is necessary to compare them to verify the operation is safe; a different rule would be needed for pointer arithmetic. In both the fourth and fifth rule, parens have been added to clearly distinguish the K = from the C assignment =.

Finally, rules six and seven show how some aspects of pointers and structures are handled. A pointer is represented as a location – a pointer to location L has the value ptr(L). On dereference, the location held in the pointer is looked up to retrieve its value. A structure is represented as a tuple containing the name of the structure type and the aforementioned finite map from field names to locations; the unneeded part of the finite map is represented as _. When field X is looked up in a structure, like S.X, the location of X is retrieved using the map and then looked up to bring back the value assigned to the field.

5.3.5 CPF[UNITS]: Annotation Language, Asserts, and Assumes

The unit annotation language is shown in Figure 9. Unit includes an operation, unit, to check the unit of an expression; unit exponents, where Q can be a rational number; basic units, such as meters or kilograms; and unit products, specifying a new unit. UnitExp includes units, tests for unit equality, and a number of logical connectives. Logical operators have their standard precedences, not reflected in the simplified grammar shown here.

The unit annotation language can be used inside annotations tagged as UNITS annotations. These annotations are changed into custom assert and assume statements, with policy-specific handling. CPF[UNITS] will check the asserts and assumes by first determining the units of any expressions, based on the current environment. assumes are then treated like unit assignments, with assignment going from right to left – unit(x) = meter assigns the unit meter to variable x, while unit(x) = unit(y) assigns the unit of y to x. By comparison, asserts are treated as logical checks, with unit comparisons performed using a combination of the units theory from Figure 4, to determine when units are equal, and the concept of unit compatibility used during unit merging when checking expressions. Unannotated functions and objects are treated conservatively, with functions given default annotations and objects assigned fresh units (described above in Section 5.3.3) that allow incorrect uses to be detected.

		Total Time			Average Per Function		
Test	LOC	x100	x400	x4000	x100	x400	x4000
straight	25	6.39	23.00	229.80	0.06	0.06	0.06
ann	27	8.62	31.27	307.54	0.09	0.08	0.08
nosplit	69	12.71	46.08	467.89	0.13	0.12	0.12
split	69	27.40	106.55	1095.34	0.27	0.27	0.27

Times in seconds. All times averaged over three runs of each test. LOC (lines of code) are per function, with 100, 400, or 4000 identical functions in a source file.

Fig. 10. CPF[UNITS] Performance

5.4 Running CPF[UNITS]

As an example of the use of CPF[UNITS], Figure 1 shows a portion of a C program that uses units. As mentioned in Section 1, it is not obvious that this code contains a unit error. By adding the annotations shown in the figure, CPF[UNITS] can check the program for unit errors. This gives the following output:

```
Function finalEnergy: ERROR on line 26(1): Assert failed!
```

This message shows that the code actually has a unit error, in this case on line 26. The potential cause of this error was explained in Section 1. Any assertion failures are similarly reported to the user, with additional information provided where this is possible. For instance, errors triggered by addition operations will report the line number and the fact that the error is caused by an invalid addition.

6 Evaluation

Evaluation was performed using two sets of experiments. All tests were performed on the same computer, a Pentium 4 3.40 GHz with 2 GB RAM running Gentoo Linux and Maude 2.3. In the first, the focus was on performance, ensuring that using a per-function analysis would scale well as desired. The results are shown in Figure 10. 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 unit annotations; `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 different environments in which statements will need to be evaluated. LOC gives the lines of code count, derived using the CCCC tool [22], for each function, with the same function repeated 100, 400, or 4000 times.

As shown in Figure 10, performance scales almost linearly: quadrupling the number of functions to check roughly quadruples the total processing time. Per-function processing time is small, making CPF[UNITS] a realistic option for checking individual functions during development, something not possible in some other solutions (such as C-UNITS) that require the entire program be checked at once. Splitting environments increases the execution time, but not prohibitively: with

Test	Prep Time	Check Time	LOC	Functions	Annotations	Errors	FP
ex18.c	0.083	0.754	18	3	10	3	0
fe.c	0.113	0.796	19	2(3)	9	1	0
coil.c	0.113	59.870	299	4(3)	14	3	3
projectile.c	0.122	0.882	31	5(2)	16	0	0
projectile-bad.c	0.121	0.866	31	5(2)	16	1	0
big0.c	0.273	5.223	2705	1	0	0	0
big1.c	0.998	22.853	11705	1	0	0	0
big2.c	33.144	381.367	96611	1	0	0	0

Times in seconds. All times averaged over three runs of each test. Function count includes annotated prototypes in parens. FP represents False Positives.

Fig. 11. CPF[UNITS] Unit Error Detection

eight environments, the time per function to process `split` is roughly double, not eight times, that to process `nosplit`, which has just one environment. Finally, processing annotations in the units annotation language seems to add little overhead; annotations are treated as statements during processing, so in some sense just add additional "hidden" lines of code.

The second set of experiments compares against some of the same examples used by Osprey, some of which were originally from C-UNITS, with the results shown in Figure 11. `fe.c` is the example shown in Figure 1; `coil.c` is part of an electrical inductance calculator; `projectile.c` calculates the launch angle for a projectile; and `projectile-bad.c` does the same, but with an intentionally-introduced unit error. `big0.c`, `big1.c`, and `big2.c` include a repeated series of arithmetic operations and are designed to test the size of function that CPF[UNITS] can handle, with `big2.c` included as an especially unrealistic example.

Overall, Figure 11 shows that the annotation burden is not heavy: assumptions on variable declarations are sometimes needed, while preconditions and postconditions are often needed, with the number of annotations needed by Osprey being similar (although those used by Osprey are generally smaller). `big0.c`, `big1.c`, and `big2.c` require no annotations, while `coil.c` requires 14, including on function prototypes. `fe.c` requires 9 annotations, with `ex18.c` requiring 10. The `projectile.c` example is particularly interesting: the use of a more flexible annotation language allows a more general version of the program to be checked than in some other systems (as discussed in Section 2), maintaining unit polymorphism, while `projectile-bad.c` includes an error not caught by Osprey, since the error involves using a variable with a different unit (pounds versus kilograms) in the same dimension. Overall, only 16 annotations are needed across 5 functions and 2 prototypes in both `projectile.c` and `projectile-bad.c`. `coil.c` shows a disadvantage of the CPF[UNITS] approach: one of the `goto` statements never stabilizes, meaning the units keep changing with each iteration. This raises an error in the program, which in this case appears to be a false positive.

7 Future Work and Conclusions

This paper presented CPF[UNITS], a static analysis tool based on an abstract rewriting logic semantics of C, designed for checking the unit safety of C programs. This tool provides a modular, scalable method of detecting unit violations. Unlike many type or library based approaches, CPF[UNITS] requires no changes to the base language, and can support relationships between the units of formal parameters, local variables, and function return values via annotations. Finally, the use of a modular framework, the C Policy Framework, and an underlying abstract semantics in rewriting logic allow for the rapid testing of new features and extensions, such as extensions to the annotation language.

There are several areas where CPF[UNITS] could be extended. First, some C code cannot yet be safely analyzed. This includes code that uses features that are not type-safe, such as pointer arithmetic and unions, as well as code that uses ambiguous function pointers. Extending the CPF[UNITS] definition, while using additional analysis information from CIL, should make it possible to safely handle more of these cases. Second, a number of conservative assumptions around aliasing and global variables preserve correctness but can generate warnings; additional analysis information from CIL should also be useful in these cases, to sharpen the analysis capabilities without losing correctness. Third, annotations on global variables and structure definitions would allow assumptions about units associated with globals or instances of structures to be stated once, instead of stating them in functions which use them; these are currently being added. Fourth, error messages are being improved. Finally, there are some useful annotations that cannot yet be properly handled, including unit annotations that depend on variables in the exponent (such as saying that, given an integer `n`, variable `x` has unit $\texttt{meter}^n$). Extending the capabilities of the annotation language would increase the power of CPF[UNITS], allowing it to handle more complex cases.

References

[1] C Policy Framework. http://fsl.cs.uiuc.edu/cpf.

[2] Mars Climate Orbiter. http://mars.jpl.nasa.gov/msp98/orbiter.

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

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

[5] 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. ACM Press, 2004.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS'04*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

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

[8] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Proceedings of FMICS'03*, volume 80 of *ENTCS*, 2003.

[9] F. Chen, G. Roşu, and R. P. Venkatesan. Rule-Based Analysis of Dimensional Safety. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 197–207. Springer, 2003.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[11] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL'77*, pages 238–252. ACM Press, 1977.

[12] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of PLDI'99*, pages 192–203. ACM Press, 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] P. N. Hilfinger. An Ada Package for Dimensional Analysis. *ACM TOPLAS*, 10(2):189–203, 1988.

[16] M. Hills, F. Chen, and G. Roşu. Pluggable Policies for C. Technical Report UIUCDCS-R-2008-2931, Department of Computer Science, University of Illinois at Urbana-Champaign, 2008.

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

[18] L. Jiang and Z. Su. Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *Proceedings of ICSE'06*, pages 262–271. ACM Press, 2006.

[19] M. Keller. EiffelUnits, 2002. http://se.inf.ethz.ch/projects/markus_keller/EiffelUnits.html.

[20] A. J. Kennedy. Relational Parametricity and Units of Measure. In *Proceedings of POPL'97*. ACM Press, 1997.

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

[22] T. Littlefair. C and C++ Code Counter. http://sourceforge.net/projects/cccc.

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

[24] 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.

[25] J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007. Also appeared in *SOS '05*, volume 156(1) of *ENTCS*, pages 27–56, 2006.

[26] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of CC'02*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[27] P. G. Neumann. Letter from the editor - risks to the public. *ACM SIGSOFT Software Engineering Notes*, 10(3):10, July 1985.

[28] G. Roşu. K: A Rewriting-Based Framework for Computations – Preliminary version. Technical Report UIUCDCS-R-2007-2926, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[29] G. Roşu and F. Chen. Certifying Measurement Unit Safety Policy. In *Proceedings of ASE'03*, pages 304 – 309. IEEE Press, 2003.