Rule-based Analysis of Dimensional Safety

Feng Chen, Grigore Roşu, Ram Prasad Venkatesan

Department of Computer Science
University of Illinois at Urbana - Champaign, USA
{fengchen,grosu,rpvenkat}@uiuc.edu

Abstract. Dimensional safety policy checking is an old topic in software analysis concerned with ensuring that programs do not violate basic principles of units of measurement. Scientific and/or navigation software is routinely dimensional and violations of measurement unit safety policies can hide significant domain-specific errors which are hard or impossible to find otherwise. Dimensional analysis of programs written in conventional programming languages is addressed in this paper. We draw general design principles for dimensional analysis tools and then discuss our prototypes, implemented by rewriting, which include both dynamic and static checkers. Our approach is based on assume/assert annotations of code which are properly interpreted by our tools and ignored by standard programming language compilers/interpreters. The output of our prototypes consists of warnings that list those expressions violating the unit safety policy. These prototypes are implemented in the rewriting system Maude, using more than 2,000 rewriting rules. This paper presents a non-trivial application of rewriting techniques to software analysis.

1 Introduction

Checking software for measurement unit consistency, also known as dimensional analysis, is an old topic in software analysis. Software developed for scientific domains, such as physics, mechanics, mathematics or applications of those, often involves units of measurement despite the lack of support provided by underlying programming languages. Computations using entities having attached physical units can be quite complex; detecting dimensional inconsistencies in such computations, for example adding or comparing meters and seconds, can reveal deep domain-specific errors which can be hard, if not impossible, to find by just analyzing programs within their language's semantics, automatically or not.

To emphasize the importance and nontrivial nature of dimensional analysis, we remind the reader two notorious real-life NASA failures. NASA's Mars Climate Orbiter spacecraft crashed into Mars' atmosphere on September 30, 1999, due to a software navigation error; the peer review findings indicate that one team used English units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation [24]. This happened less than 15 years after the space shuttle Discovery flew upside down over Maui during an experiment on June 19, 1985, in an attempt to point the mirror to a spot 10,023 *nautical miles* above see level; that number was supplied in units of *feet* and then fed into the onboard guidance system, which unfortunately was expecting units in nautical miles, not feet [22]. These two failures, as well as many other lower magnitude ones, could have been avoided by using dimensional analysis tools.

There is much work on supporting measurement units in programming languages. The earliest mention of the idea of incorporating units in programming languages, to our knowledge was in 1960 [5]. Karr and Loveman [17] suggested a flexible mechanism that allowed units to occur meaningfully in programs. There have been proposals for dimensional checking within existing languages like Pascal [9, 10] and Ada [11], and even in formal specification of software [14]. An intuitive approach to strengthen type checking in programming languages was also suggested in [16]. These approaches are based on defining a strong type checking system with physical units on top of a programming paradigm, and then using the compiler to catch dimensional inconsistencies. A similar approach was taken by the Mission Data System (MDS) team at NASA JPL, who developed a large C++ library incorporating a few hundred classes representing typical units, like `MeterSecond`, together with appropriate methods to replace the arithmetic operators when measurement unit objects are involved. However, these techniques based on type checking, in addition to adding runtime overhead due to the additional method or function calls (which can admittedly be minimized by optimized compilers), cause inconvenience to programmers, and make development of reusable software difficult. Furthermore, they limit the class of allowable (or type checkable) programs to an unacceptably low level. For example, a program calculating the geometric mean of the elements in a vector of meters needs a temporary variable which is multiplied incrementally by each element in the array; the unit of this temporary variable changes at each iteration, so it cannot be declared using any domain-specific fixed type. The solution adopted by MDS in such situations is to remove and attach types to numerical values via appropriate extractors and constructors, which, of course, is a serious safety leak.

Packages for dimensional analysis and integrity in Ada have been proposed in [15, 20], employing the use of Ada's abstraction facilities, such as operator overloading and type parameterization. Using a discipline of polymorphic programming, it was suggested in [21] that type checking can and should be supported by semantic proof and theory. This was extended in [23] using explicit type scheme annotations and type declarations and in [27] for type-indexed values in ML-like languages. The approach in [25] associated numeric types with polymorphic dimension parameters, avoiding dimension errors and unit errors. Kennedy proposed a formally verified method to incorporate, infer and check dimension types in ML-style languages [19], and provided a parametricity theorem saying that the behavior of programs is independent of the units used [18]. Thus, an abstract model of the language can be achieved to validate the equivalences of units, with the property that any program which can be type checked is unit safe. However, as Kennedy himself admitted, there are still interesting unit safe programs which cannot be type-checked, including the simple geometric mean.

All the above study based on extensions, sometimes quite heavy, of programming languages, builds a foundation for languages equipped with dimensional information. However, due to some practical, economical and/or taste reasons, these approaches have not been accepted by mainstream programmers and have not been widely used in real large applications. For instance, it is exceedingly inconvenient for programmers to rewrite a whole large application in another

language – the one extended with unit types – just to avoid measurement unit conflicts. In this paper we propose a lighter-weight rewriting-based approach to check measurement unit consistency, which has the main advantage that it does *not* modify the underlying programming language at all. The user interacts with our tools by providing code annotations, which are just comments, and by receiving safety policy violation warning reports. We provide an integrated tool containing both dynamic and static checkers, which are implemented by rewriting in Maude, and explain their trade-offs of which users should be aware.

Due to space limits and to the complexity and large size of the discussed tools, we cannot present our work in as much depth as we would like. Consequently, we will mainly focus on examples and general concepts, mentioning that our Maude rewriting implementation has more than 2,000 rewriting rules. However, we have developed a web-site dedicated to the presented work, where the interested reader can find more information (including complete source code) and download our tools, at `http://fsl.cs.uiuc.edu`. The work presented in this paper has been started by the second author as a former researcher at NASA.

2 Preliminaries

In this section we recall the basics of the BC and Maude languages. BC is the language on which we applied our measurement unit safety checking approach presented in this paper, but an implementation targeting C is under current development and one targeting Java is under design. Maude is a rewriting based executable specification language that we used to implement our prototypes.

BC Our domain-specific safety checking approach is general, both with respect to the domain of interest and with respect to the underlying programming language, but we firstly applied it to the GNU BC language, which comes with any Unix platform [1], because it is the simplest but still practical language having most of the characteristics of current imperative languages. BC [1] is an arbitrary precision calculator language, typically used for mathematical and scientific computation. The most basic element in BC is the number. BC has only two types of variables, simple variables and arrays, which are used to store numbers. The syntax of expressions and statements in BC is very similar to that of C. It includes all the arithmetic, logical and relational operators found in C, in addition to the increment "`++`" and decrement "`--`" operators, and it also allows control structures for branching and looping through constructs like `if`, `for` and `while`. Comments start with the characters `/*` and end with `*/`. BC programs are executed as the code is read, on a line by line basis; multiple instructions on a single line are allowed if separated by semicolon. It allows `auto` variables in functions, which are intended as variables for local usage; however, they are distinguished from the traditional local variables, their active range being extended over those functions called by the function in which they are defined, thus giving BC a dynamic scoping language flavor. By pushing auto variables and parameters of functions into stack dynamically, BC supports recursive functions. One can type `man bc` on any UNIX platform for more information on GNU BC. For some unexplained reason, only one-character identifiers are allowed by the BC implementation on Sun platforms; therefore we recommend the readers interested in experimenting with large examples to use the Linux version of BC.

Maude Maude [6] is a high performance specification and verification system in the OBJ family [13] that supports both equational and rewriting logics. It provides a good framework to create executable environments for different logics, theorem provers, programming languages and models of computation. We use Maude to specify the BC programming language along with its executable semantics and its domain-specific operational semantics with respect to units of measurements. Since Maude can execute its specifications efficiently, one can use these specifications to execute BC programs *directly* into the mathematical semantics of the language, and further, to analyze BC programs for safety policy violations directly into the mathematical definition of the safety policy. Therefore, Maude provides the foundation for our work in this paper. The following is a Maude specification implemented in our measurement unit safety analysis tool, defining a segment of the theory of units of measurement:

```
fmod UNITS is
  protecting INT .
  sorts BUnit SpecialUnit Unit UnitList .
  subsorts BUnit SpecialUnit < Unit < UnitList .
  ops mile kg meter second Newton Celsius Fahrenheit : -> BUnit .
  ops noUnit any fail : -> SpecialUnit .   op _^_ : Unit Int -> Unit [prec 10] .
  op __ : Unit Unit -> Unit [assoc comm prec 15] .   op nil : -> UnitList .
  op _,_ : UnitList UnitList -> UnitList [assoc id: nil] .
  vars U U' : Unit .  vars N M : Int .
  eq Newton = kg meter second ^ -2 .
  eq U noUnit = U .     eq U any = U .      eq U fail = fail .
  eq fail ^ N = fail .  eq any ^ N = any .  eq noUnit ^ N = noUnit .
  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) .
 endfm
```

We next briefly introduce the reader to the Maude notation, at the same explaining some intuitions underlying the module above needed later. The keywords `sort` and `op` refer to the types of data and the operations on these data respectively. In the above module, we have different types (sorts) of data: `BUnit` for basic units, `SpecialUnit`, `Unit` and `UnitList`. Units like `mile` and `noUnit` have been declared as constants of sorts `BUnit` and `SpecialUnit`, respectively. The intuition for the special units is that they can be used in any unit context but can be distinguished from the basic units if needed. The unit `any` is a unit which can be dynamically converted to any other unit, depending on the context; for example, in a BC statement like `x++`, the increment 1 is of unit `any` and is dynamically converted to the current unit associated to the variable `x`. The special unit `noUnit` is used to distinguish a canceled unit (for example after calculating `meter^(1-1)`) from the unit `any`, in order to report appropriate warnings, and the special unit `fail` is attached to a variable in case its unit cannot be computed due to safety violations, such as, for example, the unit of `z` after executing the BC assignment `z = x + y` in an environment in which `x` has the unit `meter` while `y` has the unit `second`. The result sort of an operation is listed after the `->` symbol, while the argument sorts are listed between the `:` and `->` symbols. The power operator defined by `op _^_ : Unit Int -> Unit` implies that the operator takes a unit and an integer and returns another unit (the power). Maude allows attributes like associativity, commutativity, precedence and identity to be associated with binary operators. This makes it a very elegant language for

specifying and manipulating sets and lists. In the above module, by virtue of `op __ : Unit Unit -> Unit` being declared commutative and associative with a precedence of 15, Maude considers `meter second` and `second meter` equivalent, which is natural. Variables of a sort can be defined using the keyword `var`. The equations in the above module, introduced via the keyword `eq`, define the semantics of the power operator. We claim, without proof, that the specification above is canonical, that is, confluent and terminating, so it can be used as a proper computational model for unit equivalences. Thus Maude provides a clean and efficient way of defining abstract domains as equational specifications.

3 Executable Semantics of Programming Languages

Equational logic is an important paradigm in computer science. It admits complete deduction and is efficiently mechanizable by rewriting: CafeOBJ [8], Maude [6] and Elan [3] are equational specification and verification systems in the OBJ [13] family that can perform millions and tens of millions of rewrites per second on standard PC platforms. It is expressive: Bergstra and Tucker [2] showed that any computable data type can be characterized by means of a finite equational specification, and Goguen and Malcolm [12], Wand [26], Broy, Wirsing and Pepper [4], and many others showed that equational logic is essentially strong enough to easily describe virtually all traditional programming language features.

Following the example of Goguen and Malcolm [12], we have defined the semantics of BC as an equational algebraic specification in Maude. Our BC specification has about 500 equations in Maude, all unconditional. Thanks to Maude's speed in executing unconditional equational specifications, we were able to run dozens of non-trivial, often recursive BC programs directly within their semantics in Maude. The overall reduction in speed was a factor of about 25-30, which we found pretty satisfactory for our prototypes, which essentially extend the executable semantics of BC with new, domain specific definitions for safety policies. Equational specifications of programming languages, as well as extensions of them, can be developed usually very rapidly because they just reflect the definition of the language. In fact, they are nothing but formal definitions of languages. We believe that this is an appropriate level to start developing a software analysis tool, because one does not have to worry about implementation details of the programming language but rather focus on the main issues. One can download our BC specification from `http://fsl.cs.uiuc.edu`, together with all its extensions to units of measurement, and soon those for C and Java.

Briefly, the executable semantics of BC declares an operation `run` which takes a program and a list of integers (its input) and returns another list of integers (its output). To execute programs properly, one needs to also define environments, which are sets of pairs (variable, integer). Because of recursive function calls, one needs to also stack these environments dynamically, as the program executes. Appropriate operations to update the environment are also defined, as well as operations to properly deal with return, break and continue statements.

4 Design Conventions and Annotation Schemas

The design of our prototypes has been influenced by three major factors: correctness, unchanged native programming language, and low amount of annotations.

Correctness By "correctness" we mean that there are no possible violations of safety policy in the program which our tool does not report. We consider correctness a crucial aspect because, unlike other tools like Extended Static Checking [7] being mainly intended to *help* users find some bugs in their programs with relatively little effort, our tools are intended to be used in the context of safety critical software, such as air craft and navigation, where software engineers, our users, want to be aware of any inconsistency in their code.

Unmodified Programming Language Another major influencing factor in the design of our prototypes was the decision to *not* modify the underlying programming language at all, for example by adding new types. Our reason for this decision is multiple. First, we do not want to worry about providing domain specific compilers; one can just use the state of the art optimized compilers for the specific programming language under consideration, without any modification. Second, by enforcing an auxiliary typing policy on top of a programming language in order to detect unit inconsistencies via type checking, one should pay the price of some runtime overhead due to method calls that will replace all the normal arithmetic operators; our static prototype does not add any runtime overhead. Third and perhaps most importantly, since we do not add new types to the language, we do not put the user in the unfortunate situation to have a correct program rejected because it cannot be type checked, which is in our view the major drawback of typed approaches to unit safety; instead, our user has the option to either add auxiliary unit specific information to help the checker or to ignore some of the warning messages.

Annotation Schemas The mechanism by which the users can add auxiliary, in this case measurement unit specific, information to their program is by means of annotations, which are inserted as special comments at appropriate places in the program. Annotations are introduced with the syntax `/*U _ U*/`, which is understood by our tools and ignored by compilers, and are of two kinds: assumptions and assertions. Our annotation schemas are general and can be applied to any domain-specific safety policy checker, but in this paper we will focus on unit safety policy.

Assumptions. There are two types of assumptions currently supported by our tools, namely `/*U assume unit(_) = _ U*/` and `/*U assume returns _ U*/`. The first can appear anywhere in the program and takes as arguments (the underscores) a variable and a unit expression. The variable can be an indexed one in the dynamic version and should be a simple one in the static version; if it is not a simple one then it will be automatically replaced by its simple root, e.g., `s[10][i]` will be replaced by just `s`. The unit expression can be any combination of basic units and `unit(Expr)`, the second being evaluated in the current execution environment(s). For example, if `x`, `y`, `acc`, `dist` and `time` are variables then the following are admissible assumptions:

```
/*U assume unit(x) = meter U*/    /*U assume unit(x) = unit(y) U*/
/*U assume unit(acc) = meter second^-2 U*/
/*U assume unit(acc) = unit(dist/time) second^-1 U*/
```

The second assumption annotation is used only for functions, to enforce the unit of the returned value to a specific unit. It can be used within unit conversion functions, such as from Fahrenheit to Celsius (we will show such an example

soon), or simply to state the result unit of a function when that cannot or is not desired to (in order to keep the complexity of the tool low), be inferred. It is placed just before the body of the function and takes a unit expression as a sole argument, which will be evaluated before the body of the function but after the arguments of the function are instantiated. For example, the following naive function calculates the square root over integers and can be applied on any units:

```
define sqrtNaive(x) /*U assume returns unit(x)^(1/2) U*/
{ auto temp;  temp = 0;
  while (1) {if (temp*temp == x) return temp; if (temp*temp > x) return temp - 1; temp += 1;}
}
```

In fact, our tools will report warnings for the two conditionals above because they cannot prove that the units of `temp*temp` and `x` are compatible. However, the returned value of `sqrtNaive` will be used as `unit(x)^(1/2)` in every callee context, where the unit of `x` is calculated dynamically, also in the callee's context.

Assertions. Assertions have the syntax `/*U assert _ U*/`, where the argument can be any boolean expression on units, using the usual connectors `and`, `or`, `implies`, `not`, over equalities of unit expressions. The next are some examples:

```
/*U assert unit(x) = meter U*/    /*U assert unit(x) = unit(y) U*/
/*U assert unit(acc * time ^ 2) = unit(dist) and unit(speed) unit(time) = unit(dist) U*/
/*U assert unit(x) = unit(y) implies unit(z) = unit(x) ^ 2 U*/
```

It is worth noticing that our assertions can be highly unit invariant. For example, the variables above can be represented either in the metric or in the English system as far as they respect the specified dimensional constraints. Assertions can be anywhere in a program, including just before the body of a function:

```
define Fah2Cel(z)/*U assert unit(z) = Fahrenheit U*//*U assume returns Celsius U*/
{ return (z - 32) * 5 / 9; }
```

Example. We next present a less trivial example showing some of the complex unit expressions that can be manipulated by our tool, and also emphasizing the importance of annotations. The program below provides functions to calculate distances, convert energy and calculate the angle under which a projectile of a given weight should be launched in order to travel a given distance:

```
define sqrtnaive(x) {
  auto temp;  temp = 0;   /*U assume unit(temp) = sqrt(unit(x)) U*/
  while(1) {if (temp*temp>=x) return temp; if (temp*temp>x) return temp-1; temp += 1;}
}
define lb2kg(w) /*U assert unit(w) = lb U*/ /*U assume returns kg U*/
{return 10 * w / 22;}
define distance(x1, y1, x2, y2) {return sqrtnaive((x2-x1)^2 + (y2-y1)^2);}
define energy2speed(energy, weight) {return sqrtnaive(2 * energy / weight);}
define prjTangent(dist, speed, g)  /*U assert unit(speed)^2 = unit(dist) unit(g) U*/
{ auto dx, dy;
  dx = speed * speed + sqrtnaive(speed^4 - (dist * g)^2); dy = dist * g
  return dx/dy
}
projectilex = 0;       /*U assume unit(projectilex) = meter U*/
projectiley = 0;       /*U assume unit(projectiley) = unit(projectilex) U*/
targetx = 17;          /*U assume unit(targetx) = unit(projectilex) U*/
targety = 21;          /*U assume unit(targety) = unit(projectiley) U*/
dist = distance(projectilex, projectiley, targetx, targety);
projectileweight = 5;  /*U assume unit(projectileweight) = lb U*/
energy = 2560;         /*U assume unit(energy) = kg meter^2 second^-2 U*/
speed = energy2speed(energy, projectileweight);
g = 10;                /*U assume unit(g) = meter second^-2 U*/
print(prjTangent(dist, speed, g));
```

The first function is the naive implementation of square root and the second one is a converting routine, from `lb` to `Kg`. The next function computes the distance between two points. No annotations are given, but a warning will be generated anyway if the arguments do not have the same unit. The fourth function computes the speed of an object, given the energy acting on it. The last function computes the tangent of the angle of a projectile, given a certain distance it wants to reach, an initial speed and a gravitational acceleration. This function is annotated with an assertion describing a unit invariant among its arguments. This allows one to use such functions in various contexts, such as under metric or English system conventions, as well as for other possible combinations of units. One can now assume a context in which these functions are called. The above code contains a unit safety violation, which is immediately reported by both checkers. The error will be reported when the function `projectileTangentAngle` is called, because the unit of speed is `Kg^(1/2) meter second^-1 lb^(-1/2)` so the assertion of function `projectileTangentAngle` will be violated. To correct this problem, the user should first properly convert the projectile weight to `Kg`, so the speed should be assigned the expression `energy2speed(energy, lb2kg(projectileWeight))`.

Reducing the Amount of Annotations Another major factor influencing our design significantly was the overall observed and sometimes openly declared reluctance of ordinary programmers and software engineers to modify or insert annotations in their programs. Therefore, we paid special attention to reducing the amount of annotations to a minimum possible. As a consequence, every variable by default is considered to have its own unit, which is different from any other existing unit. This principle of course extends to auto variables, their units being considered different from the units of global variables having the same name. Our tool will therefore output a warning on the simple program `print(x+y)`, because `x` and `y` cannot be shown to have the same unit. Similarly, the program `x = 10; y = 10; print(x+y)` will be output a warning on its third statement, for the same reason.

This brings us to another major design convention of our tools, which we call the *locality principle*, which says that the user is assumed to know and understand what (s)he is doing locally, within a single instruction, with respect to constants. For example, if one writes `x++`, then one means to increase the value of `x` by 1, and this 1 has exactly the same unit as `x` at that particular moment during the execution of the program. The same increment instruction can be reached twice or more times during the execution of the program; each time the unit of the increment will be dynamically converted to the unit of `x`, which can be different each time. There is, and there should be, no difference between the statements `x++` and `x = x + 1`, so we apply the same locality principle to numerical constants. That implies that in `x = x + 5`, the unit of 5 will be converted to the unit of `x` and no warning will be reported. Additionally, a constant assignment to a variable, such as `x = 5`, will not change the unit of `x`. Our motivation for these conventions is again to keep the amount of code annotation low, but the users thinking that our locality principle yields a safety leak have the option to always attach a unit to numerical values via appropriate assumptions,

e.g., `temp = 5 ; /*U assume unit(temp) = second U*/`, and then execute `x = x + temp`; a warning will be reported in this case if the unit of `x` cannot be shown to be `second`. Based on these efforts, the following example of sorting needs only one assumption to satisfy the safety policy:

```
n = 25 ;  /*U assume unit(i) = any U*/
for (i = 1 ; i <= n ; i = i + 1) a[i] = n - i + 1 ;
for (i = 1 ; i < n ; i = i + 1)
  for (j = i + 1 ; j <= n ; j = j + 1)
    if (a[j] < a[i]) { temp = a[i] ; a[i] = a[j] ; a[j] = temp ; }
for (i = 1 ; i <= n ; i = i + 1) print(a[i]) ;
```

The only assumption needed, assigning the universal unit `any` to the counter `i`, guarantees the compatibility of `i` and `n` when they are compared later, within the loop conditions. The first loop, which assigns decreasing numbers to the elements of the array `a`, also assigns the unit of `n`, which is considered to be a fresh unit different from any other unit because no unit has been explicitly assigned to it, to each of the 25 elements of `a`. In the case of the static analyzer, the array `a` will be assigned the unit of `n` by executing the loop body symbolically only twice, regardless of the value of `n` (because the environment set stabilizes; see Subsection 5). Then the second loop is analyzed and no warning is reported because the nested loop assigns the unit of `i`, which is `any`, to `j`, so any subsequent comparisons of `j` are safe; the environment set also stabilizes in two iterations of the loop. Similarly the third loop can be certified without any auxiliary information. Without the assumption, 5 warnings would be reported.

5 Rule-based Dynamic and Static Analysis Tool

We next present our tool and its underlying algorithms. We start by describing how one interacts with our tool, mentioning that the interested reader can download from `http://fsl.cs.uiuc.edu`. The current version of our tool supports only the BC language, but it is being extended to support C and Java.

Tool Development The tool is invoked with the command "`bc-unit [-so] filename`" where `s` and `o` are optional. By default, the tool starts its dynamic checker, which is described below, and its output consists of a list of warnings reported in the order of execution. A warning consists of the line number and the expression violating safety; the same warning can appear many times if the unsafe expression has been executed more than once. The option "`-o`" tells the tool to output the warning list ordered by line numbers without redundancy. This option decreases the amount of reported safety violations, so only advanced users should use it. The option "`-s`" triggers the static checking mode, which is also described below.

The user of our tool should understand the differences and the tradeoffs between dynamic and static safety policy checkers. The main advantage of the dynamic checker is the precision of its reported warnings: any reported warning represents a violation of the unit safety policy. The user should therefore consider these reports very seriously and should have strong reasons to ignore them. The main drawback of the dynamic checker is its coverage: it only covers the path that was traversed by the particular execution of the program. Therefore, other errors might exist in the analyzed program which were not revealed and which can appear when the program is executed with different numerical values as

input. Another drawback of the dynamic unit safety checker is that its execution time consists of the analyzed program execution time plus the runtime overhead. Therefore, if a program calculates a computationally complex function or does not terminate in a reasonable time, then so does the unit safety prototype, which can be a serious drawback in some applications.

An advantage of the static checker discussed next is that it covers all the potentially reachable code. So it will not miss any unsafe expression. A careful and patient analysis of the reported warnings can lead one to find all the unit safety leaks. Another advantage is its relative efficiency, because it does not execute the programs, so non-termination of the program does not imply non-termination of the tool. However, depending on the amount of theorem proving that one wants to put in such a static certifier, it can actually become rather inefficient. A major drawback of the static certifier is the potentially long list of false alarms that it reports. The user can reduce their number using assume annotations, but one should be careful when using assumptions because they can be wrong and thus present a serious safety threat.

We used Maude as a rule-based programming language to implement both the dynamic and the static checkers. Since programs to be analyzed are expected to be provided as terms to Maude, we have implemented a simple wrapper (about 300 lines of PERL code), whose work-flow is the following:

1. Adjust the input program to be parsable by Maude, e.g., add spaces around BC symbols, add line numbers, delete unnecessary comments, detect the variable and function names and define them as appropriate constants, etc.;
2. Invoke either the dynamic or the static Maude checker (which are described below) to verify the generated program term;
3. Collect Maude's output, parse it and produce user friendly error messages.

This way users can use our tool in a push-button fashion, without seeing Maude.

Dynamic Checker Our Maude dynamic unit safety checker essentially interprets the BC program within its executable semantics enriched with the unit safety policy. This is realized by extending the executable semantics specification of BC discussed in Section 3. A major extension is with respect to execution environments. An execution environment is now a set of triples, each triple containing a variable, an integer value and a unit of measurement. The integer value is used to determine the execution flow of the analyzed program, while the unit is used to check the safety policy. For example, if the expression `x + y` is encountered at line number 15 and the execution environment contains the triples `[x, 7, meter^2 second]` and `[y, 3, meter second]` then the value 10 is correctly assigned to the sum of `x` and `y` but a warning message will be issued of the form `15 : x + y`. If the expression `x + y` was assigned to a variable, say `x`, then the new environment will assign the value 10 and the unit `fail` to the variable `x`. Notice that, since BC supports recursive function calls and auto variables, all the environment stacking technique needs to be extended to the enriched environments.

Another major extension of the BC semantics is with respect to the newly introduced code annotations, which act like new, domain-specific instructions. One has to describe their semantics also in an executable manner. An assumption

`/*U assume unit(Var) = UnitExp U*/` is interpreted by our dynamic checker as follows: 1) first evaluate the unit expression `UnitExp` in the current environment, hereby obtaining a result which is an expression using just basic units, then 2) modify the current environment by associating the newly calculated unit to the variable `Var`, without changing its current integer value; if `UnitExp` fails to evaluate to a correct unit, due to violations of the safety policy that it may contain, then the unit `fail` will be assigned to `Var`. Due to its precision in analysis because of the exact execution path and environment, the dynamic checker can allow the user to assign different units to different elements in an array. In fact, any abstract memory location (which can store an integer; note that BC is arbitrary precision) can have any unit associated with it. Assumptions `/*U assume returns UnitExp U*/` are interpreted as follows: when a function call is invoked in an expression context, `UnitExp` is evaluated and returned as unit associated to the function call; the function call will also take place, and all additional warnings while executing the function are collected and output to the user. Assertions of boolean unit expressions are simply evaluated to boolean values and warnings are returned if they evaluate to false. Assertions act as just checks in the context of dynamic checking.

The Maude implementation of our current unit safety dynamic checker uses a total of about 1,000 implicit or explicit rewriting or membership rules.

Static Checker The main idea behind our static unit analyzer is that the concrete execution path of the program to be checked is entirely ignored; instead, all execution flows are considered in parallel. An immediate simplifying abstraction assumption is to also ignore all the numerical values (integers) and only consider the domain-specific, abstract values (units of measurement) of variables. Therefore, our environments will now consist of pairs (variable, unit), rather than triples (variable, integer, unit) like in the case of dynamic unit safety checking. Moreover, since the concrete indexes in arrays (integers) are not available anymore we'll have to also assume that, unlike its dynamic version, all the elements in an array have the same unit.

Due to the loss of precision, at each point in the program we will have to consider a *set* of environments, namely all those in which a potential execution of the program can be. The necessity of this can probably be seen best in the case of conditionals. If one wants to analyze a program of the form `if (E) Stmt else Stmt' ; Pgm` in an environment `Env` then, due to the fact that the concrete values of variables are not known, one basically has to consider both branches, followed by `Pgm`. If one first considers the first branch followed by `Pgm` and then the second followed by `Pgm`, then one misses the whole point which makes unit certification relatively tractable in practice, namely that despite the apparent exponential branching of environments, these will actually repeat often. In particular, if the two branches generate the same environment when they terminate (before starting `Pgm`), then the approach above would check the remaining program twice, which is undesirable. The situation is actually even worse, because evaluating an expression can generate an arbitrary number of possible environments (due to function calls). Instead, we prefer to work with sets of environments in parallel. Each statement will be abstractly evaluated in all the environments. If the unit

safety policy will be violated in any of the environments then a warning will be output. A new set of environments will be computed after each statement. Set operations, such as idempotency, will be executed on the fly, so the set of environments is kept small.

An additional complication is the treatment of return statements. Consider for example the conditional above and that one of the branches contains a return statement. Then we must be able to "freeze" the set of environments that encounter the return statement, and to return those (after they are properly popped) to the callee's environment when all the function's code was processed. Any subsequent code will be of course discarded by frozen environments.

Another complication is the treatment of loops. A general solution would involve loop invariants, which we would like to avoid as much as possible due to its lack of understanding by ordinary programmers and software engineers. Our alternative solution, which seems quite promising and very suitable for and actually inspired from rule-based programming, is one based on *code patterns*. More precisely, we define loop patterns that we can efficiently analyze statically. One such pattern is a loop whose body, when symbolically executed under a set of possible environments, does not change that set of environments; if this is the case then the loop can be safely ignored. A generalization of this pattern which we have also implemented is one which symbolically executes the loop until the set of environments stabilizes; this pattern for example is triggered in order to analyze the sorting algorithm in Subsection 4. Other more complex patterns are also considered. If a certain loop does not fall under any of the provided patterns, then all the defined variables in the loop are invalidated (their unit is set to `fail`) and the static analysis process continues. The user can intervene and attach proper units to failed variables. The advanced user can add new patterns.

The Maude implementation of our current static checker for unit safety uses a total of about 1200 Maude rewriting rules.

Some Experiments In most of our experiments the Maude executable definition of BC was about 25-30 times slower than BC v1.06 on Linux platforms, which is a good factor considering that we build our tools directly on top of a mathematically clean setting. Adding measurement unit knowledge to the BC specification basically doubles its size (adds 500 more axioms) and further increases the execution time of programs by a factor of 5. Since many physics programs essentially calculate a function and have just one execution path, one can argue that having a precise dynamic measurement unit safety checker is extremely useful, even if it slows down the execution of the program by 2-3 orders of magnitude. The static checker needs more axioms (rules) because it implements several patterns for loops. If a loop falls under a known pattern then it is discarded quickly; otherwise, the user intervention may be needed to add new assumptions as annotations. The killing complexity factor for the static checker is conditional branches when these modify the unit-specific environment differently, which in our experience happens very rarely. We tried examples where 10 worst-case 10 line conditionals were serialized, for a total of 1024 situations to analyze, and it took our static checker about 3 seconds to check it. On the other hand, if the two branches generate the same abstract environment, that is, if

they both modify the units of variables in the same way, then we were able to analyze 1,000 repetitions of best-case 10 line conditionals, so a total of 10,000 lines of BC code, in about 1 second.

6 Conclusion and Future Work

A promising annotation based approach to dimensional safety has been presented, together with a tool including both dynamic and static safety checkers implemented by rewriting in Maude. Future work includes extending the pattern database for the static checker and designing a general purpose invariant based loop analysis technique to be launched as a last pattern before the defined variables are invalidated. This work is supported by a joint NSF/NASA grant, CCR-0234524, which assumes that the supported projects will be run on NASA testbeds and used by NASA scientists.

References

1. GNU BC. URL: `http://www.gnu.org/software/bc/bc.html`.
2. J. Bergstra and J.V. Tucker. Equational specifications, complete rewriting systems, and computable and semicomputable algebras. *JACM*, 42(6):1194–1230, 1995.
3. P. Borovanský, H. Cîrstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN. User manual – http://www.loria.fr.
4. M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Trans. on Prog. Lang. and Systems*, 9(1):54–99, January 1987.
5. T. Cheatham. Handling fractions and n-tuples in algebraic languages. Presented at the 15th ACM Annual Meeting, Aug. 1960.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, http://maude.csl.sri.com.
7. Compaq. ESC for Java, 2000. URL: `www.research.compaq.com/SRC/esc`.
8. R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, volume 6.
9. A. Dreiheller, M. Moerschbacher, and B. Mohr. Physcal - programming Pascal with physical units. *ACM SIGPLAN Notices*, 21(12):114–123, December 1986.
10. N. Gehani. Units of measure as a data attribute. *Comp. Lang.*, 2:93–111, 1977.
11. N. H. Gehani. Ada's derived types and units of measure. *Software: Practice and Experience*, 15(6):555–569, June 1985.
12. J. Goguen and G. Malcolm. *Alg. Semantics of Imperative Programs*. MIT, 1996.
13. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.
14. I. J. Hayes and B. P. Mahony. Units of measurement in formal specifications. Technical report, SVR Centre, University of Queensland, November 1994.
15. P. N. Hilfinger. An ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, April 1988.
16. R. T. House. A proposal for the extended form of type checking of expressions. *The Computer Journal*, 26(4):366–374, 1983.
17. M. Karr and D. B. Loveman III. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, May 1978.

18. 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.
19. A. J. Kennedy. *Programming Languages and Dimensions*. PhD thesis, St. Catherine's College, University of Cambridge, November 1995.
20. G. W. Macpherson. A reusable ada package for scientific dimensional integrity. *ACM Ada Letters*, XVI(3):56–69, 1996.
21. R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.
22. Peter G. Neumann. Letter from the editor - risks to the public. *ACM SIGSOFT Software Engineering Notes*, 10(3):10, July 1985.
23. M. Odersky and K. Läufer. Putting type annotations to work (preliminary). Technical report, Newton Institute Workshop on Advances in Type Systems for Computing, Cambridge , England, Ausust 1995.
24. Mars Climate Orbiter. URL: `http://mars.jpl.nasa.gov/msp98/orbiter`.
25. M. Rittri. Dimensional inference under polymorphic recursion. In *Functional Programming Languages and Computer Architecture, 7th Conference*. Association of Computing Machinery, Inc, 1995.
26. M. Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980.
27. Z. Yang. Encoding types in ML-like languages. In *ICPF 98*. Association of Computing Machinery, Inc, 1998. Baltimore, USA.