Automatic and Precise Dimensional Analysis

Marcelo d’Amorim, Mark Hills, Feng Chen, and Grigore Roșu

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801,
{damorim, mhills, fengchen, grosu}@cs.uiuc.edu

Abstract. The loss of NASA’s Mars climate orbiter [5] is evidence of the
importance of units of measurement as a safety policy for software in gen-
eral and for scientific applications in particular. In this paper we present
a static analysis technique that detects violations of the unit policy. The
technique relies on domain-specific unit annotations inserted in the code,
either manually or automatically with the support of a tool, which are
verified conservatively, i.e., all runtime unit errors are detected statically
using an automatic theorem prover. This paper informally compares our
approach with others, describes the technique in detail, and evaluates a
benchmark built of standard programs for unit analysis and important
fragments of NASA’s SCRover project code.

keywords: units of measure, predicate transformers, static analysis.

1 Introduction

Units of measurement are employed in various fields of science not only to assign
measures to quantities, but especially to constrain the axioms of mathematics
within which these quantities operate. In this paper we use (measurement) units
as a domain of analysis in order to frame the set of program executions inside a
safety envelope consisting of basic domain-specific unit consistency requirements.
We will refer to dimension as a domain classifying measurable values, e.g., dis-
tance, time, speed, weight, etc. A unit corresponds to a particular dimension and
makes it possible to measure quantities. For instance, meter is a particular unit
in the distance dimension, while second is a unit of time. This paper describes a
tool-supported technique for the static analysis of units of measurement, often
called dimensional analysis.

We recall the importance and non-trivial nature of dimensional analysis with
two notorious real-life failures. NASA’s Mars climate orbiter spacecraft crashed
into Mars’ atmosphere on 30 September 1999 due to a software navigation error.
Peer review indicated that one team used the English unit system (e.g., inches,
feet, etc.) while the other used the metric system for a key spacecraft operation
[5]. On 19 June 1985, the space shuttle Discovery flew upside down over Maui
during an experiment in an attempt to point the mirror to a spot 10,023 feet
above sea level. This number was supplied to the onboard guidance system, which
unfortunately was expecting units in nautical miles [33]. These two failures and
possibly many others could have been avoided by using dimensional analysis
tools like the one presented in this paper.

* Supported in part by the joint NSF/NASA grant CCR 0234524, NSF CCF-0448501,
and NSF CNS-0509321.
Overview. Our technique relies on the generation of verification conditions (VCs) using the standard weakest-precondition semantics [23, 15, 21]. In addition to constraints on actual values of the program, the VCs we generate also contain constraints on the units associated to program variables. The combination of concrete and abstract domains in the VC is made necessary because of the existence of operators producing new units, in particular the operators for multiplication and division, whose evaluation can become assigned to variables in the program. For this reason, the verification condition can contain units of the form \((\textbf{unit } x)^{exp}\), where \textbf{unit } x denotes the unit of a variable in the program and exp an expression in the language. For instance, the geometric mean procedure, \((\textbf{gmean})\) in Figure 1, requires an invariant \textbf{unit } \texttt{tmp} == (\textbf{unit } x)^j \land \ldots\), where \texttt{tmp} is the temporary computing the product, \(x\) is the input array, and \(j\) the loop induction variable. Proofs are carried out modulo the axioms of units using an \textbf{automatic} (non-interactive) [34, 14] theorem-prover.

It is worth noting that the user is \textit{not} forced to annotate every program loop with unit assertions. This is the case, for instance, when the variables in a loop do not change units (which is the case for all our previous attempts [12, 36]). These pre-defined patterns as well as “simple” unit constraints can be automatically generated in a processing step that runs before the analysis. As pointed out in [12, 36] this is not sufficient to show the previous example unit safe, but is enough for checking unit safety without annotations on a class of programs, including all those that could be analyzed by our previous techniques. The following example shows the generation of unit constraints by a pre-processor for the assignment \(x = a \ast (c + d)\). The operands of an addition must agree in their units while a multiplication generates a unit which is the product of the units of the operands. This new unit becomes “assigned” to the unit of \(x\). The unit safety of arithmetic operators is then expressed via annotations \textit{generated automatically} in the program. The command \textbf{havoc} assigns an arbitrary unit to an identifier. Note that we translate (also automatically) the program into 3-address code to facilitate the generation of annotations.

```plaintext
let int x1, int x2 in {
  x1 = a ; havoc unit(x1) ; assume unit(x1) = unit(a) ;
  assert unit(c) = unit(d) ;
  x2 = c + d ; havoc unit(x2) ; assume unit(x2) = unit(c) ;
  x = x1 * x2 ; havoc unit (x) ; assume unit(x) = unit(x1) unit(x2) ;
}
```

Related Work. Numeric types associate to polymorphic dimension parameters in [35], avoiding dimension and unit errors. A formal verification method is proposed in [25] to incorporate, infer and check dimension types in an extension of ML, and, in [26], a parametricity theorem is given stating that the behavior of a program is independent of the units used. In our work, we explore the possibility, unfulfilled by type-checker based approaches, that programs might \textit{change units of variables and still be unit safe}. A simple example of the limitation of typed approaches is the calculation of the geometric mean of the sequence of values \(\{a_i\}_{i=1}^n\) given by \((\prod_{i=1}^n a_i)^{1/n}\). A program that computes the mean
usually defines a temporary variable to accumulate the intermediate values of the product. Assuming all values in the sequence have unit \( u \), the unit of the temporary variable will therefore change \( n - 1 \) times before the \( n \)-th root of the temporary is taken, leaving a final value with unit \( u \). A type-checker assigns a single, possibly parametric, unit, say \( u \), to variables and therefore will reject the possibility of the temporary in the example above having units \( u, u^2, \ldots, u^n \). Our technique is capable of analyzing such admittedly useful programs and thus will not reject them as dimensionally inconsistent.

Frameworks can assist in the consistent use of units via libraries of frequently used units with basic arithmetic operations and conversions between them. Such a lightweight approach can be practical for simple applications with a fixed number of units, however, it requires significant programming effort because all arithmetic operations need to be performed via procedure calls. Unfortunately, libraries can only support a fixed number of units (as their associated types are statically defined), which is inconvenient. The assumption that only a fixed number of units are needed can be misleading since a simple multiply and assign statement such as \( x = x \times y \) can change the unit associated to \( x \).

Yet another approach is to make use of the existing type system of a language by translating accesses to variables of primitive types annotated with units into accesses to objects whose classes correspond to units, i.e., primitive values are boxed inside special unit objects. Two works in this category are [4, 8], where a new language providing support for dimensions and units is introduced. This approach has the advantage in comparison to the framework-based approach in the automation of code generation for enforcing the consistent use of unit operations. However, [4] suffers from the fact that only a finite number of units can be used in programs – 300 “common” units are defined as classes with several hundred arithmetic operations between them in methods. While [8] allows an infinite number of units, it cannot cope with the possible change of units at runtime without over-generalizing the unit type, which causes a loss of information about the exact unit type of an object. Also, [8] requires an entirely new language, making it more challenging to examine existing applications for unit policy safety violations. It is important to mention that despite the fact that we use a simplified language for demonstrating our technique, we do not pose any language requirement (we rely on annotations which are ignored by compilers).

We have previously defined a measurement unit static checker for units of BC [12] and C [36] programs using symbolic execution. Similarly to type-checking [25] it reports spurious warnings when proving facts that involve variables whose units change as in the geometric mean example. In this paper, we introduce a tool-supported technique that is able to detect all unit violations that our previous attempts detected and yet allows variables to change units consistently. In particular, we will be able to analyze the geometric mean example correctly, that is, without reporting any violation.

Some Background. We begin by showing the basic algebraic properties of units. The algebraic structure of units forms an abelian group:
An abelian group is comprised of an operator and a set of elements, and satisfies the following properties for any of its elements \(A, B, C\).

- **Closure:** the product \(AB\) is a group element.
- **Associativity:** \(A(BC) = (AB)C\).
- **Identity:** there is an identity element \(I\) s.t. \(AI = IA\).
- **Inverse:** every element \(A\) admits an inverse \(A^{-1}\) s.t. \(AA^{-1} = I\).
- **Commutativity:** for any two group elements \(A\) and \(B\), \(AB = BA\).

Note that the product of two units, here denoted by concatenation, gives a different unit (e.g. meter seconds is a derived unit from meter and second), the order of units in a product is irrelevant, there exists a unit identity – from hereon called \(\text{noUnit}\) – and for any unit \(u\) we can always calculate its inverse \(u^{-1}\) because we have a power operator. Furthermore, the product of two units is commutative (e.g. meter seconds = seconds meter) which finally illustrates that the unit algebra comprised of product and power operators, the special unit \(\text{noUnit}\), and a set of unit constants indeed forms an abelian group. It is known that unification in an abelian group is a decidable problem \([10]\). Kennedy \([26]\) then showed a type-checker for ML augmented with units. We explore the fact that the algebra of units forms an abelian group, but take a different approach than type-inference/checking to assure unit safety of programs.

This paper is organized as follows. Section 2 describes the analysis of units, Section 3 presents an initial implementation, Section 4 shows the evaluation of our implementation and Section 5 concludes.

## 2 The Analysis of Units

In contrast to type-checker \([35, 26]\) and framework based \([22, 29, 4]\) techniques our analysis allows variables to change units multiple times via assignment as long as the assigned expressions remain unit consistent along the execution of the program. We believe that programs whose variables change are quite common and not considering this will lead to a significant number of false alarms. Furthermore, in contrast to the techniques mentioned above and also to those based on symbolic execution \([12, 36]\) we also take into account the concrete values associated to program variables in order to give more precision to our analysis.

As exposed in \([12, 36]\) using symbolic execution, it is often the case that local unit assertions are not sufficient to show the entire program unit safe. Consider, for instance, that a variable, say \(x\), has its unit changed inside a loop. A unit violation must be raised in an expression such as \(x + E\) occurring after the loop if \(x\) and \(E\) cannot be proved to have the same unit. As usual, safety of loops needs to be characterized by the existence of fix-points, and unit invariants may be required to prove their correctness. In the geometric mean example, we know, partially from the invariant, that the unit of tmp after the loop is the unit of \(x\) raised to the power of \(\text{length}\) and this is a necessary condition to calling the root procedure and then proving the post-condition of gmean.

Unit annotations can be introduced manually but their generation is also partially automated. The user might need to define unit invariants in order to prove a particular procedure unit safe, but the tool also generates unit constraints from the semantics of the program and the algebraic structure of units in order
to check for local unit misuse. These pre-defined annotations together with a conservative pattern for calculating the fix-point of loops [36] might suffice to prove several programs unit safe [12, 36] and substantially reduce the amount of annotation needed to analyze a program.

The technique we present in this paper is concerned with the generation of a first-order predicate \( Q \) derived from a procedure with the property that if \( Q \) is proved to be valid under certain assumptions then that procedure does not contain unit errors. Otherwise, a violation could have happened and we issue a warning pointing out the place in the code that might have caused the violation. In contrast to approaches like ESC/JAVA which compromises soundness with efficiency [17, page 235], we generate verification conditions conservatively in order to avoid missing true alarms. We pass a proposition of the form \((P \land BG) \rightarrow Q\) as a proof task to the prover, where \( Q \) is a verification condition derived from a procedure involving not only constraints on the units of variables but also on the concrete program values and characterizes the initial program state, \( BG \) is a background predicate axiomatizing the abelian group algebra in the particular domain of units, and \( P \) is the precondition of the analyzed procedure. The proposition informally means that if the precondition holds then it must be the case that \( Q \) must hold under the additional assumption of \( BG \). Refutation-based theorem provers, like Simplify [14], return a counter-example \( C \) with the following properties [28]: \( C \) is satisfiable, and \( C \rightarrow \neg((P \land BG) \rightarrow Q) \). That is, \( C \) expresses a way to refute the predicate passed to the prover. Refutation of a proof task therefore corresponds to a safety violation. When a formula such as \( C \) is found, the tool can give meaningful feedback to the user.

We first present the command language we use in our analysis and its dynamic semantics. We follow by presenting the algorithms to generate annotations and to calculate VCs, showing the axioms of units which form the background predicate for verification, and the loop patterns which are useful for loops defined without an invariant. We show in a soundness theorem that the technique will not miss real unit runtime errors. In the next section, we discuss an initial implementation of these ideas which makes use of the BOOGIE verifier [13]. Boogie acts as our interface to the automatic theorem prover SIMPLIFY [14], and handles some of the tasks needed for verification, including generation of verification conditions and translation of the Simplify output into useful error messages.

2.1 The Core Language (CL)

Table 1 describes the EBNF of a simple proof-of-concept command language whose programs are the objects of our analysis. It is important to mention that we do not intend to introduce a language expressive enough to enable the translation of many real programming languages to it, but rather to expose a simple language for demonstrating our technique. We believe that this language includes an important subset of features for conveying evidence that the presented technique can be extended to modern languages. In this section we also describe informally the dynamic semantics of this “core” language, CL for short.

Expressions of this language are side-effect free and assumed to be given in 3-address form [6]. The language has integers, arrays of integers, and booleans as
denoted values. We assume that procedures come with pre and post-conditions and anticipate that the analysis proceeds in a modular fashion, looking at one procedure at a time. Therefore, function calls are translated to assert and assume annotations in the callee site and recursion will not become an issue.

The symbol “+,” appearing in the exponent of terms denotes one or more occurrence, separated by commas, of the elements in the mentioned syntactic category. Placeholders appear underlined, and terminals in bold. The following are the commands of the language: sequential composition, assignment to a simple variable and to an array element, non-deterministic choice, arbitrary assignment, local variable declaration, assertion, and assumption. A program is a comma-separated sequence of procedures (Proc). The keyword result denotes the return value of a procedure.

Variables are declared without initialization via a scoping command let. false is assumed as the initial value for booleans, 0 for integers, and null for arrays. The expression new int[id] allocates an array of size id. The only operations involving arrays are assignment of an array to a variable, assignment of an array element, and selection. CL has a call-by-value semantics even for arrays.

| Proc ::= requires: AExp ensures: AExp Type proc Id(TypedId+) { Com } |
| Com ::= Com ; Com | Id = Exp | Id [Id] = Id | Com □ Com | havoc (Id) |
| Type ::= int | int[] | boolean | Unit |
| TypedId ::= Type Id |
| Id ::= “program identifiers” |
| Exp ::= “int. constants” | Unit | Id | Id[Id] | new int[Id] | Id α Id, α ∈ {+, −∗, /} |
| AExp ::= AExp pop AExp | qu TId . AExp | ¬ AExp | (AExp) | Atom |

Table 1. The EBNF for the core language.

The type Unit becomes necessary in order to quantify over the units of program variables. The set of expressions includes integer literals annotated with units, identifiers, array selection, array allocation, and arithmetic expressions.

The set of assertion expressions (AExp) is built from first-order formulae. The placeholder pop stands for one of the propositional operators, and qu for a first-order quantifier.

Extended Assertion Language. We extend the assertion language of first order formulae on program variables with unit equality.

**Definition 2.** The set of atoms is defined as follows:

\[ Atom ::= \text{true} | \text{false} | Exp \rho Exp | \text{Unit} = \text{Unit} \]

where \(\rho\) ranges over \(\text{==, !=, >, >=, <, <=}\).

**Definition 3.** The set of units is defined as follows:

\[ Unit ::= \text{noUnit} | \text{“constant unit”} | \text{Unit} | \text{Unit} \uparrow (Exp) | \text{unit}(Id) \]
where the set of constant units includes *meter, second, newton*, and so forth; concatenation denotes the product of two units, ^ is the unit power operator, and unit is an uninterpreted function denoting the unit of a variable. Note that the power of a unit takes a program expression as argument. This construct can express relationships between units and concrete program values, e.g. the proposition \( \text{unit}(x)^i = \text{unit}(y)^i \) relating the unit of \( x \), the unit of \( y \), and the concrete value stored in the variable \( i \).

From hereon we overload the command havoc for the uninterpreted function unit since the units of variables will also be treated “as” program variables. In addition, \( \text{havoc}(id_1...id_n) \) is a shorthand for \( \text{havoc}(id_1);...;\text{havoc}(id_n) \) and \( \text{havoc}(\text{unit}(id_1)...\text{unit}(id_n)) \) a shorthand for \( \text{havoc}(\text{unit}(id_1));...;\text{havoc}(\text{unit}(id_n)) \).

**Desugaring.** Loops, branching, and procedure calls are not expressed in the core language. They belong to a language extension of CL and are considered syntactic sugar. A loop of the form \( \text{inv: } P \text{ while } b \text{ do } \{ c \} \) translates to:

\[
\begin{align*}
&\text{assert } (P) ; \\
&(\text{havoc}(ids) ; \text{havoc}(\text{unit}(ids))) ; \text{assume } (P \land b) ; c ; \text{assert } (P) ; \text{assume}(\text{false}) \\
&\quad \lor (\text{havoc}(ids) ; \text{havoc}(\text{unit}(ids))) ; \text{assume } (P \land \neg b)
\end{align*}
\]

We first check that the invariant holds in the loop entry. As we do not know in the loop entry or exit the values assigned to the identifiers (ids) in the body, we explicitly “discard” them with the command havoc. The first block in the choice assumes the invariant and the loop guard and then checks the invariant after the execution of the body. We do not need any further check from this path if the assertion is valid. On the other hand, the assumption made in the alternative command on the choice will get propagated to the commands that will follow.

A conditional of the form \( \text{if } b \text{ then } \{ c \} \text{ else } \{ c' \} \) translates to \( \text{assume}(b) ; c \lor \text{assume}(\neg b) ; c' \), and a function call of the form \( \text{call } id = id'(ids) \) translates to \( \text{assert}(Pre) ; \text{havoc}(id) ; \text{assume}(Post) \) where \( Pre \) and \( Post \) are the pre and post-conditions declared in the function named \( id' \) and have occurrences of formals properly substituted by actual parameters. In addition, we translate language expressions and assertion expressions of the enriched language into the 3-address form of CL in order to simplify the definitions that will follow in this paper. Booleans are expressed in the language to allow the translation of assertion expressions into 3-address form.

**Example.** The procedure gmean in Figure 1 calculates the geometric mean of a sequence of values. It takes an array and the array length as parameters, calculates the product of the sequence and then calls a second procedure, root, to compute the root of the product. Note that the procedure root plays only the role of an interface, i.e. its verification will be trivially satisfied. As usual in assume-guarantee reasoning, the pre and post-conditions of the called procedure are taken into account to analyze the callee code.

This program is written in our CL extension. After going through a pre-processing step consisting of the translation of control-flow constructs, the analyzer instruments unit annotations, generates verification conditions, and passes a predicate derived from the VC to an automatic theorem-prover to be analyzed.
requires: length > 0 & unit(length) == noUnit
ensures: unit(result) == unit(x)
int proc gmean (int[] x, int length) {
  let int j, int tmp in {
    tmp = 1 ;
    inv: unit(tmp) == unit(x)^j & unit(j) == noUnit & j <= length
    while (j < length) do {
      tmp = tmp * x[j] ; j = j + 1 }
    result = call root(tmp, length)
  }
},
requires: unit(n) == noUnit & EXISTS Unit k . unit(pow) == k ^ n
ensures: unit(result) == unit(pow) ^ (1 / n)
int proc root (int pow, int n) { assume(false) }

Fig. 1. The geometric mean program.

2.2 Dynamic Semantics and Unit Errors
As a static bug-finding technique our purpose is to detect runtime errors at compile-time. The dynamic semantics of CL characterizes the runtime behavior of its programs including unit errors. Here we briefly introduce the dynamic semantics of CL, ignoring errors such as “array access out of bounds” that do not relate to units. Later we show, in a soundness theorem, that the errors the analyzer finds properly include the dynamic errors.

We present the semantics in a big-step style with two relations: $ʃ_{e}$ for expressions, and $ʃ_{c}$ for commands. A concrete environment $Σ_c$ maps identifiers to concrete values $V$, while a unit environment $Σ_u$ maps identifiers to their units $U$. Maps support an update operation $[\cdot] ← [\cdot]$, and $\emptyset_c$ and $\emptyset_u$ denote initial (empty) concrete and unit environments, respectively. The arrow on expressions relates a triple comprised of an expression, a concrete environment, and a unit environment with a value-unit pair representing the result of evaluating the expression. We use the symbol $error$ to denote an evaluation error in either relation without risk of confusion, i.e. $error$ belongs to the codomain of both relations. The axioms below express the correct and incorrect evaluation of addition and subtraction. The symbol $α'$ ranges over $\{+,-\}$:

$\langle id_1 \alpha' id_2, σ_v, σ_u \rangle \Downarrow_{e} (σ_v(id_1) \alpha' σ_v(id_2), σ_u(id_1)) \quad σ_u(id_1) = σ_u(id_2)$

$\langle id_1 \alpha' id_2, σ_v, σ_u \rangle \Downarrow_{e} error \quad σ_u(id_1) \neq σ_u(id_2)$

The arrow on commands relates a triple comprised of a command, concrete environment and unit environment with a tuple of concrete environment and unit environment, denoting a new compound state. The satisfaction relation on assertion expressions, denoted $|=\quad$, takes the form $σ_v, σ_u |= P$ and denotes the satisfaction of $P$ under concrete and unit environments $σ_v, σ_u$. The rules below describe a correct assignment and an incorrect command sequence.

$\langle e, σ_v, σ_u \rangle \Downarrow_{c} (i, u)$
$\langle id = e, σ_v, σ_u \rangle \Downarrow_{c} (σ_v[id ← i], σ_u[id ← u])$  
$\langle c_1, σ_v, σ_u \rangle \Downarrow_{c} error$  
$\langle c_2, σ_v, σ_u \rangle \Downarrow_{c} error$

Appendix A details the semantics.
2.3 Annotation Generation

This section describes the generation of unit annotations for “core” CL programs. We perform a simple context-insensitive transformation to include the necessary annotations for checking unit safety. Table 2 describes this annotation generation algorithm via the command transformer \( ag : Com \to Com \). Note that, thanks to the 3-address form, arithmetic operations only appear in assignments and occur with at most one operator. This is also the case for relational expressions built with the operator \( \rho \) derived from assertion expressions and conditionals. The \texttt{havoc} command [17,13] appearing in Table 2 will override the previous value (unit, resp.) of the mentioned variable with an arbitrary one. Therefore, one can consider “as assigned” a value (unit) given by the \texttt{assume} command that follows.

2.4 VC Generation

Figure 2 defines the axiomatic semantic of CL with the predicate transformer \( wp_{\text{Com}} : AExp \to AExp \) calculating the weakest precondition of each command in \( Com \). This transformer gives an algorithm to generate verification conditions of programs. The term \( \{id_1; id_2 : id_3\} \) denotes an array \( id_1 \) with the expression \( id_3 \) assigned to the element indexed by \( id_2 \). The substitution operator \( [\cdot]_Q \) for assertion expressions replaces every free occurrence of

\[
\begin{array}{ll}
  c & ag(c) \\
  c_1 ; c_2 & ag(c_1) ; ag(c_2) \\
  id = i\{u\} & id = i \;\; \text{havoc}(\text{unit}(id)) \;\; \text{assume}(\text{unit}(id) == u) \\
  id_1 = id_2 & id_1 = id_2 \;\; \text{havoc}(\text{unit}(id_1)) \\
    & \;\; \text{assume}(\text{unit}(id_1) == \text{unit}(id_2)) \\
  id_1 = id_2 \;\; \sigma \;\; id_3, & \;\; \text{assert}(\text{unit}(id_2) == \text{unit}(id_3)) \;\; \text{havoc}(\text{unit}(id_1)) \;\; \text{assume}(\text{unit}(id_1) == \text{unit}(id_2)) \\
    & \;\; \sigma \in \{+, -\} \\
  id_1 = id_2 \;\; \rho \;\; id_3, & \;\; \text{assert}(\text{unit}(id_2) == \text{unit}(id_3)) \;\; \text{havoc}(\text{unit}(id_1)) \;\; \text{assume}(\text{unit}(id_1) == \text{unit}(id_2)) \\
    & \;\; \rho \in \{+, -\} \\
  id_1 = id_2 * id_3 & id_1 = id_2 * id_3 \;\; \text{havoc}(\text{unit}(id_1)) \\
    & \;\; \text{assume}(\text{unit}(id_1) == \text{unit}(id_2) \;\; \text{unit}(id_3)) \\
  id_1 = id_2 / id_3 & id_1 = id_2 / id_3 \;\; \text{havoc}(\text{unit}(id_1)) \\
    & \;\; \text{assume}(\text{unit}(id_1) == \text{unit}(id_2) \;\; (\text{unit}(id_3) \;\; \cdot \;\; (-1))) \\
  id_1[id_2] = id_3 & \;\; \text{assert}(\text{unit}(id_2) == \text{noUnit}) \\
    & \;\; \text{assert}(\text{unit}(id_1) == \text{unit}(id_3)) \;\; \text{havoc}(id_1[id_2] = id_3) \\
  c_1 \;\; \square \;\; c_2 & ag(c_1) \;\; \square \;\; ag(c_2) \\
\text{let} \;\; \text{type} \;\; id \;\; \text{in} \;\; \{c_1\} & \text{let} \;\; \text{type} \;\; id \;\; \text{in} \;\; \{ag(c_1)\} \\
\text{havoc}(id) & \text{havoc}(id) \\
\text{havoc}(\text{unit}(id)) & \text{havoc}(\text{unit}(id)) \\
\text{assume}(P) & \text{assume}(P) \\
\text{assume}(P) & \text{assume}(P)
\end{array}
\]

Table 2. The annotation generation transformer.

\[\begin{array}{ll}
  c & wp(c, Q) \\
  c_1 ; c_2 & wp(c_1, wp(c_2, Q)) \\
  id = e & Q[\text{id} \;\; \setminus \;\; e] \\
  id_i[id_2] = id_3 & Q[\text{id}_1 \;\; \setminus \;\; \{\text{id}_1; \;\; \text{id}_2 : \;\; \text{id}_3\}] \\
  c_1 \;\; \square \;\; c_2 & wp(c_1, Q) \;\; \land \;\; wp(c_2, Q) \\
\text{let} \;\; \text{type} \;\; id & \text{FORALL} \;\; \text{id}: \text{type}, \;\; \text{id}^* : \text{Unit} . \\
\text{in} \;\; \{c\} & wp(c[\text{unit}(\text{id}) \;\; \setminus \;\; \text{id}^*], Q) \\
\text{havoc}(id) & Q[\text{id} \;\; \setminus \;\; \text{id}^*] \;\; \land \;\; \text{id}^* == k \\
\text{assert} \;\; P & P \;\; \land \;\; Q \\
\text{assume} \;\; P & P \;\; \Rightarrow \;\; Q
\end{array}\]

Fig. 2. The VC generator.
an identifier in a predicate by a given term. The symbols \text{$id^*$} and $k$ stand for fresh names. The fresh variable $k$ represents an arbitrary value (or unit). The fresh name $k$ in the definition of \texttt{havoc} will appear in the VC as a schematic variable. Note that \texttt{havoc(id)} : \texttt{assume (id == \ldots)} is not equivalent to \texttt{assume (id == \ldots)} as the proposition $id == \ldots$ can be falsified in some proof state, making the path under verification become trivially satisfied. Datatypes for integers, finite maps, for arrays, and units are assumed in the object language.

2.5 The Axioms of Units

Figure 3 presents an axiomatization of the unit algebra. It lists several axioms stating the equivalence of units. Note from the second axiom that \texttt{noUnit} is an identity in the group. The last two axioms state associativity and commutativity properties. One alternative way to define these axioms is using an already defined theory of groups. We preferred to use this stand-alone theory instead mainly because not every TP supports a theory for groups, e.g. Boo-\texttt{gie/Simplify} [13]. We use typed first-order logic as our object language. This set of unit axioms together with the definition of the datatype \texttt{Unit} and its constant inhabitants form a background predicate, $BG$, which is necessary to carry out unit verification [26, 12, 36, 8]. As one might expect the set of base individuals inhabiting the type \texttt{Unit}, e.g. \texttt{meter}, \texttt{second}, \texttt{degree}, etc., is fixed but can be extended without requiring modification of the axioms of $BG$.

After generating unit annotations with the transformer $ag$, one can calculate a verification condition of a procedure using the predicate transformer $wp$. These VCs are further augmented with the background predicate stated in Figure 3, properly formatted to the syntax of a specific TP, and passed to an automatic theorem prover for verification. We assume as given a theory of finite maps and integer arithmetic including equalities and inequalities as these are required to prove the predicates derived from CL programs. The following theorem shows that the analysis of commands are sound w.r.t. the dynamic semantics.

Soundness Theorem 1 For any CL command $c$, concrete environments $\sigma_v$, $\sigma'_v$, unit environments $\sigma_u$, $\sigma'_u$, and predicate $Q$, $\langle c, \sigma_v, \sigma_u \rangle \Downarrow_c \langle \sigma'_v, \sigma'_u \rangle$ and $\sigma_v, \sigma'_u \models BG \Rightarrow Q$ if $\sigma_v, \sigma_u \models BG \Rightarrow wp(ag(c),Q)$.

Proof. By structural induction on the syntax of $Com$. Take $c$ to be $c_1; c_2$ then from the definitions of $ag$ and $wp \sigma_v, \sigma_u \models BG \Rightarrow wp(ag(c_1),wp(ag(c_2),Q))$. As $wp$ is a total function on $Com$, take $Q'$ to be $wp(ag(c_2),Q)$ then $\sigma_v, \sigma_u \models BG \Rightarrow wp(ag(c_1),Q')$, and by hypothesis $\langle c_1, \sigma_v, \sigma_u \rangle \Downarrow_c \langle \sigma'_v, \sigma'_u \rangle$ and $\sigma'_v, \sigma'_u \models BG \Rightarrow Q'$, i.e. $\sigma'_v, \sigma'_u \models wp(ag(c_2),Q)$, which establishes from the hypothesis that $\langle c_2, \sigma'_v, \sigma'_u \rangle \Downarrow_c \langle \sigma''_v, \sigma''_u \rangle$ and that $\sigma''_v, \sigma''_u \models BG \Rightarrow Q$. It follows from the sequential composition rule (see Appendix A, Table 4) that $\langle c_1; c_2, \sigma_v, \sigma_u \rangle \Downarrow_c \langle \sigma''_v, \sigma''_u \rangle$. The other commands can be proved similarly.
The corollary that follows shows that our static analyzer does not miss unit errors. Informally it says that if one can prove the VC generated for some procedure valid under the assumption that the precondition holds and that the unit axioms given are valid, then it should not be the case that any unit error will be raised during the execution of the analyzed procedure.

**Corollary 1** For any CL procedure with body \( c \), pre-condition \( Pre \) and post-condition \( Post \), any concrete environment \( \sigma_v \), and unit environment \( \sigma_u \), \( \langle c, \sigma_v, \sigma_u \rangle \not\downarrow \text{error} \) and \( \sigma_v, \sigma_u \models Pre \) if \( \models Pre \land BG \Rightarrow wp(ag(c), Post) \).

**Proof.** This follows as a special case of the Soundness Theorem. Assume valid the predicate \( Pre \land BG \Rightarrow wp(ag(c), Post) \) then one can discharge the assumption \( Pre \) into a state \( \sigma_v, \sigma_u \) s.t. \( \sigma_v, \sigma_u \models Pre \) and \( \sigma_v, \sigma_u \models BG \Rightarrow wp(ag(c), Post) \). Therefore, from the previous theorem, there exists some compound state \( \sigma'_v, \sigma'_u \) s.t. \( \langle c, \sigma_v, \sigma_u \rangle \Downarrow \langle \sigma'_v, \sigma'_u \rangle \) and \( \sigma'_v, \sigma'_u \models Post \). Then error is unreachable.

### 2.6 Loop Patterns

We introduce loop patterns as a means to avoid the requirement for user-defined invariants in every loop by considering the possibility of the units remaining constant across iterations. We believe that loops having this property are common and a simple invariant like this should cover a large number of procedures. A loop of the form \( \text{while } b \text{ do } \{ c \} \) translates to:

\[
\text{let } t_1 \ id_1^*, ..., t_n \ id_n^* \{ \text{assume}(P) ; \text{inv: } P \text{ while } (b) \text{ do } \{ c \} \}
\]

where \( P \) stands for the expression \( \text{unit}(id_1^*) == \text{unit}(id_1) \land ... \land \text{unit}(id_n^*) \) == \( \text{unit}(id_n) \), and \( id_1, ..., id_n \) are the variables assigned in the loop body. Recall that we do not have side-effects so this information can be easily collected. We define “old versions” for these variables to save their associated units, i.e. the unit of \( id_1^* \) will become the unit of \( id_1 \) prior to the loop entry. The invariant then states that no change in the unit of modified variables will occur in the body. Therefore this is a safe approximation of the unknown unit loop invariant. In the case units in fact change, an alarm will be raised allowing the user to redefine the assumed but incorrect invariant.

### 3 Verification Implementation

We showed how to translate a program with unit annotations into a first order formula stating unit safety. We illustrate next how these programs can be verified in our initial implementation of the above concepts, which makes use of the BOOGIE verifier [13] which currently uses the refutation-based theorem-prover SIMPLIFY. BOOGIE relies on its own specification language and therefore makes our task of generating VCs unnecessary.

**The BOOGIE (Spec#) Verifier.** BOOGIE [13] not only generates verification conditions efficiently [18] but also checks safety of programs modulo some user-defined axiomatic theory. The tool accepts a high-level typed imperative program (specification) in the BOOGIEPL language, generates VCs for each procedure implementation, and calls the SIMPLIFY theorem prover [14] in order to check for safety violations. The tool provides command-line options to output the
verification conditions in the efficient [18] form of passive commands, i.e. without assignments, which could be possibly formatted for use in other theorem provers. BOOGIE has been used as the verifier for SPEC# and MSIL [11]. It has built-in theories for finite maps and integer arithmetic. In addition, it is open to the axiomatization of new theories. The possibility of defining the algebra of units in a high-level language within an environment integrated with the theorem prover, and having the VCs generated efficiently were key factors for choosing this tool to implement our technique. We show informally next the translation of CL programs into BoogiePL.

Translating CL into BoogiePL. The axioms of units described in Figure 3 appear in the proof task associated with every translated CL program and augment the set of theories supported by the tool, which already includes finite maps (necessary to handle arrays), partial orders (necessary to express constraints on the subtyping relation), and integer arithmetic. Units are represented as finite maps from base units to unit exponents, allowing each unit to be represented in a canonical form. The BoogiePL language omits the choice command, and includes procedure calls and labeled blocks together with a jump instruction. For the variable $id$, we call $id_u$ the variable of type $Unit$ implicitly associated with $id$. It means that every declaration of a name, either locally or as a formal parameter, spawns the declaration of a corresponding unit “variable”. The signature of the procedure $gmean$, for instance, will be extended to take four parameters, the original two plus their units. The declaration of a local variable such as $let \ id \ in \ \{ c \}$ will translate to $var \ id^* : int , \ id^*_u : unit ; \ \tau(c[id|id^*])$ where $\tau$ is a compiler taking sentences from CL to BoogiePL, and $var$ is the local declaration statement in BoogiePL. The command $c_1 \ □ c_2$ translates to $goto \ L1 , \ L2 ; \ L1 : \ \tau(c_1) ; \ goto \ L3 ; \ L2 : \ \tau(c_1) ; \ goto \ L3 ; \ L3 : \ \tau(c_1)$ where $L1, L2$ and $L3$ are fresh identifiers denoting labels in the program. The dynamic semantics of $goto$ is to randomly choose one flow of control among those informed.

4 Evaluation

Our analysis goes through a pre-processing step of desugaring, annotation generation of the input CL program, and translation to a BoogiePL program. Then, using the Boogie verifier, a first-order logic formula is built and passed to the theorem-prover SIMPLIFY. The first step consists of a sequence of program transformations defined using the equational logic [30,31] fragment of the high-performance rewriting system of MAUDE [2].

We show in Figure 4 the time spent with verification of BoogiePL programs with and without unit errors. We use Boogie shipped under the SPEC# distribution version 0.6. We tested our programs on a Pentium M 1.73GHz with 512Mb RAM running WindowsXP Pro. The programs in this benchmark appear in Appendix C. The program projectile appears in [12,36] to determine the angle to launch a projectile in order for it to hit a target. The program takes as input weight, initial position, position of the target, speed, energy, and gravity. An incorrect unit is assigned to the weight of the projectile leading to a unit
error in the procedure **main**. This error is found by the analyzer which was able to identify the contract violated. The second and the third entry only consider the time to analyze the procedure **main** on the projectile code. The fourth entry show the time to analyze the entire projectile code. We have been able to analyze the insertion sort procedure containing nested loops where the units of variables remain unchanged.

<table>
<thead>
<tr>
<th>Program</th>
<th>error</th>
<th>TP time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gmean</td>
<td>N</td>
<td>0.061000</td>
</tr>
<tr>
<td>projectile.main</td>
<td>Y</td>
<td>0.031250</td>
</tr>
<tr>
<td>projectile.main</td>
<td>N</td>
<td>0.046875</td>
</tr>
<tr>
<td>projectile</td>
<td>N</td>
<td>0.281250</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>N</td>
<td>0.921875</td>
</tr>
<tr>
<td>scrover1</td>
<td>Y</td>
<td>0.015000</td>
</tr>
<tr>
<td>scrover2</td>
<td>Y</td>
<td>0.015000</td>
</tr>
</tbody>
</table>

**Fig. 4.** Benchmark showing theorem proving time in seconds.

**The SCROVER case.** SCROVER is a robot-controlling system developed at the University of Southern California, using the Mission Data Systems (MDS) framework which NASA JPL (Jet Propulsion Laboratory) will use in their Rocky 7 rover to be sent to Mars in the year 2009. In order to avoid system errors caused by units of measurement, the MDS framework provides a large library of classes and constants to support units and their conversions. The MDS framework does not perform a static analysis on the program to catch unit errors. Instead, it relies on a few auxiliary classes to perform consistent operations involving units.

Two out of three bugs which have been publicly reported in SCROVER [4] have been caused by the misuse of units of measurement. The following is a report on the first unit bug: “I assumed the atan function returned values in degrees when it actually returned values in radians. Thus, the rover’s target turn angle is higher than it should be” [4]. The second error is reported as: “During the execution of a turn, the rover does a core dump. The problem is that we are trying to compare a number vs. a number with a SI unit” [4].

We translated the functions that produced these errors, appearing as `scrover1` and `scrover2` in Figure 4, into CL. Some irrelevant details were removed in the translation, e.g. output and command issuing functions. Our analyzer was able to detect both bugs. The first error is caused by a contract violation in the use of a procedure. It is caught with a simple **assert** annotation placed in the return of a call. The second is caused by the misuse of unit types and can be caught with the instrumented unit assertions.

Our prototype implementation can be downloaded from our website [1].

## 5 Conclusions and Future Work

Motivated mainly by the extensive use of physical units in the development of scientific applications, this paper describes a static analysis to check unit consistency in programs built from a proof-of-concept language. Our technique starts by instrumenting the program with annotations corresponding to assumptions that must be made for operations to be considered unit consistent. It then calculates, using a standard weakest-precondition semantics, a first-order predicate which includes propositions involving not only the values associated to program

---

1 The International System of Units[3].
variables but also their units. This gives more precision to our analysis at the
expense of requiring theorem provers to support a larger set of theories related
to the operations on concrete values of the program. We rely on automatic the-
orem provers equipped with those theories and demonstrate our technique with
the BOOGIE [13] analyzer against a benchmark including procedures known to
be difficult to show unit safe and fragments of the NASA’s SCROVER program,
which reports two unit errors.

In contrast to other approaches, this technique allows units associated to
variables to change. The geometric mean program is an example of a unit safe
procedure whose variables change units. In this technique, users can give unit
invariants to loops, therefore allowing one to show, in contrast to others [12],
that the geometric mean program is in fact unit safe. If invariants are not given
our tool will assume that units attached to variables remain unchanged across
loop iterations. This constitutes a conservative invariant.

We used the efficient rewriting system of MADE to implement our program
transformations. MADE itself has an interactive theorem prover (ITP) [2] which
could possibly be tailored to be used as an automatic procedure for proving our
proof obligations if provided with specialized tactics and domain-specific rewrite
rules. Another possible direction would be to enrich automatic theorem provers
with an interface to MADE so that equational theories like that of BG (see
Figure 3) could be used efficiently by plugging functional modules into the TP.

Finally, we assume in the paper that invariants are given by the user when
needed, but we plan to use abstract interpretation techniques [9] and/or random
testing [16] to detect domain-specific invariants.

Acknowledgments

We thank Darko Marinov for his comments on earlier versions of this paper.

References

7. E. Allen, J. Bannet, and R. Cartwright. A First-Class Approach to Genericity. In
8. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and J. Guy L. Steele. Object-
Oriented Units of Measurement. In OOPSLA ’04, pages 384–403, New York, NY,

Appendix A. Dynamic Semantics

We present the dynamic semantics in a big-step style with two relations: $\downarrow_e$ for expressions, and $\downarrow_c$ for commands. A concrete environment $\Sigma_v$ maps identifiers to concrete values $\mathbb{V}$, while a unit environment $\Sigma_u$ maps identifiers to their units $\mathbb{U}$. The arrow on expressions relates a triple comprised of an expression, a concrete environment, and a unit environment with a tuple of a value and its associated unit, which together denote the evaluation of an expression. Table 3 presents the relation $\downarrow_e \subseteq \langle \text{Exp}, \Sigma_v, \Sigma_u \rangle \times \langle \mathbb{V}, \mathbb{U} \rangle$ giving the semantics of CL expressions. The arrow on commands relates a triple comprised of a command, concrete environment and unit environment with a tuple of concrete environment and unit environment, denoting the new compound state. Table 4 shows the transition relation $\downarrow_c \subseteq \langle \text{Com}, \Sigma_v, \Sigma_u \rangle \times \langle \Sigma_v, \Sigma_u \rangle$.

Maps are supported by update $\left\langle j \rightarrow v \right\rangle$ and selection $\left( j \right)$ operations taking the map as the first argument and having the usual behavior. We use the symbol error to denote an evaluation error in either relation without risk of confusion, $\emptyset_v$ and $\emptyset_u$ to characterize the initial states of the concrete and unit environment maps, and $\top$ to denote both arbitrary values and units. We use the satisfaction relation $\models$ on assertion expressions, taking the form $\sigma_v, \sigma_u \models P$, to denote satisfaction of $P$ under concrete and unit environments $\sigma_v, \sigma_u$. Furthermore, we have a substitution operator $\left[ \cdot \right]$ for commands with the expected semantics.

We use the symbols $n, i, i_0, i_1, \ldots, i_n$ to denote variables for integer literals and values without risk of confusion, $id, id_1, \ldots, id_n$ to denote program variables, $e, e_1, \ldots, e_n$ expression variables, $c, c_1, \ldots, c_n$ to denote command variables, and $P, Q$ to denote assertion expressions variables. The notation $n : \langle 0 : i_0, \ldots, n - 1 : i_{n-1} \rangle$ represents an array value of size $n$. The selection operator $\left( \cdot \right)$ is overloaded for arrays. It expects an array value and an integer index within the array range with noUnit attached. This operator associates to the left.

Table 5 presents the dynamic semantics of commands w.r.t. unit errors. Informally, error propagates via sequential composition, any assignment whose expression raises a unit error will also raise an error, an error is raised in a non-deterministic choice if any compound command raises an error, and finally assume and assert have similar dynamic behavior.

Appendix B. More Related Work

Extended Static Checker for Java (ESC/Java) [17] is a static analyzer tailored to find not only common bugs such as array out of bounds and null dereference, i.e. runtime errors, but also to check user-defined assertions and object invariants defined in a syntax closely related to the Java Modeling Language (JML) [27]. The tool calculates verification conditions using the weakest precondition semantics of a simple intermediate language to which Java programs are translated. Each VC generated corresponds to a Java class method which can be annotated with pre- and post-conditions, loop invariants, and assertions. The VCs are generated so as to avoid exponential blow up in their size [18], traditionally caused by the
The transition relation for CL expressions.

Table 3. The transition relation for CL expressions.

The transition relation for CL commands.

Table 4. The transition relation for CL commands.
assignment and choice commands, and are passed to the Simplify [14] automatic theorem prover (TP). If the prover is able to prove the conjecture passed to it, then no bug of the kinds checked have been detected. The system is unsound and incomplete, i.e. only a subset of real bugs may be found and the proof can be falsified without the existence of actual bugs. As we demonstrate in the paper our technique is similar to the one employed in ESC/Java in many aspects. Our novel contribution is the extension of an assertion language with unit constraints and the definition of an analyzer, which depends both on the abstract as well as the concrete domain of values, as a simple extension of the verification conditions generated from the axiomatic semantics of a programming language.

Proof Carrying Code (PCC) [32] is a technique aimed at assuring correctness of external code to be run in some runtime environment. The technique collects proof obligations from the code with respect to some safety policy, which is comprised of rules and procedure interfaces. These verification conditions are assembled with the code and must ascertain its correctness. The host system uses a simple and efficient oracle (a decision-procedure in the VC object logic) to check the validity of the proof shipped with the program to be run. The soundness of the proof system assures that if the proof is valid than the external code is guaranteed to execute without violating the policy. Our system follows a similar approach where the safety policy is drawn from the axioms of units of measurement. However, our major purpose is not certification of mobile code.

MetaGen [8], an extension of the MixGen [7] extension of Java, provides languages features which allow the specification of dimension and unit information for object-oriented programs. While distinctive in being a solution targeting object-oriented languages with nominal typing, it follows in the footsteps of a number of other approaches making use of language and type system extensions, such as work on ML [25], Pascal [19, 24], and Ada [20]. In our case, we do not extend either the language or the type system, instead making use of annotations to indicate unit constraints.

Table 5. The transition relation for CL commands leading to error.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c_1, \sigma_v, \sigma_u) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(c_1; c_2, \sigma_v, \sigma_u) ⇓ (\psi_e) error</td>
<td>Sequential command leading to error</td>
</tr>
<tr>
<td>(\langle id_1, id_2 \rangle, \sigma_v, \sigma_u) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(\langle c_1, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Choice command leading to error</td>
</tr>
<tr>
<td>(\langle c_1; c_2, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Sequential command leading to error</td>
</tr>
<tr>
<td>(\langle id_1, id_2 \rangle = \langle id_3, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(\langle c_1, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(\langle c_1; c_2, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Sequential command leading to error</td>
</tr>
<tr>
<td>(\langle c_1, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(\langle e, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(\langle id_1, id_2 \rangle = \langle id_3, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(\langle c_1, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Assignment command leading to error</td>
</tr>
<tr>
<td>(\langle c_1; c_2, \sigma_v, \sigma_u\rangle) ⇓ (\psi_e) error</td>
<td>Sequential command leading to error</td>
</tr>
</tbody>
</table>

\(P\) is a safety property that the code is required to satisfy. The analyzer checks that the code is correct with respect to the specified property. The host system uses a simple and efficient oracle (a decision-procedure in the VC object logic) to check the validity of the proof shipped with the program to be run. The soundness of the proof system assures that if the proof is valid than the external code is guaranteed to execute without violating the property. Our system follows a similar approach where the safety policy is drawn from the axioms of units of measurement. However, our major purpose is not certification of mobile code.