Certifying Domain-Specific Policies

Michael Lowry[a], Thomas Pressburger[a], Grigore Roşu[b]

[a]Automated Software Engineering Group
[b]Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, California, 94035-1000, USA
`http://ase.arc.nasa.gov/{lowry,ttp,grosu}`

Abstract

Proof-checking code for compliance to safety policies potentially enables a product-oriented approach to certain aspects of software certification. To date, previous research has focused on generic, low-level programming-language properties such as memory type safety. In this paper we consider proof-checking higher-level domain-specific properties for compliance to safety policies. The paper first describes a framework related to abstract interpretation in which compliance to a class of certification policies can be efficiently calculated. Membership equational logic is shown to provide a rich logic for carrying out such calculations, including partiality, for certification. The architecture for a domain-specific certifier is described, followed by an implemented case study. The case study considers consistency of abstract variable attributes in code that performs geometric calculations in Aerospace systems.

1. Introduction

The benefits of product-oriented approaches to certification as compared to process-oriented approaches is to enable more efficient software development processes to be used as technology progresses, while at the same time providing higher levels of assurance by focusing safety concerns directly on the product rather than the process by which it is developed. Techniques like theorem proving or model checking have been formulated in research labs for product-oriented program verification. These techniques have high computational complexity. Static analysis has good scaling properties, and has been proposed for certifying limited types of safety, such as absence of arithmetic or memory type safety. In this paper we describe a technology for certification of domain-specific properties.

In this paper we propose a technique based on abstract interpretation that is designed to find domain-specific inconsistencies rather than programming-language specific errors. This method automates finding errors that would normally take intensive review by a human expert. Moreover, this technique is relatively easy to implement and modular, so it is easy to adapt to various domains. Thus even though the range of applicability for any particular domain-specific property is more limited than low-level programming language errors, we believe it will still be cost-effective.

Conceptually, the knowledge used by a *domain-specific certifier* has two distinct levels: an *abstract domain* specification and a *programming language* specification. These levels are linked via two modules: a *symbolic evaluation* module and a *domain-specific safety abstraction* module. The symbolic evaluation module simulates the execution of the program, yielding for each variable at each statement a term denoting the functional value of the variable in terms of functions and input variables. The domain-specific abstraction module takes such terms and interprets them into the abstract domain. A program is *domain-specific safe* if and only if each value calculated along its execution path is safe.

Our domain-specific certification approach requires more sophisticated reasoning than in approaches to date for proof-carrying code [13]. The abstract domain specification is much richer than memory safety, and verifying the safety of each line of code can require hundreds of inference steps in membership equational logic. Nonetheless, these calculations are bounded (with caching) for each value along an execution path. The two specification levels are also independently reusable; e.g., once an abstract domain has been formulated it can be used to certify programs written in various programming languages, and conversely, programs can be certified for various domain-specific safety policies.

Our method uses membership equational logic as implemented in Maude for both domain and programming language specifications. This logic extends both order-sorted and partial equational logics. Efficient algorithms for first-order rewriting and type inference (more precisely: least-sort computation) are implemented in Maude.

Section 2 describes the architecture of a domain-specific certifier and then an implemented case study: certifying the *frame*-safety of programs that calculate observation geometries. Our abstract domain knowledge describes the conditions under which calculations involving matrices and vectors are consistent with respect to coordinate frames, which are attributes of variables in the abstract domain but only implicit in the program. The calculations are performed using subroutine calls from a library developed by the NAIF group at JPL. Section 3 then introduces the technology we used: the technical notions of membership equational logic used in performing the symbolic calculations, and the Maude system. Section 4 describes the abstract domain. Section 5 describes related work, and Section 6 presents our conclusions, scaling issues, and future work.

2. Architecture of a Domain Specific Certifier

This section first presents general principles and components of a domain specific safety policy certifier and then briefly introduces our problem of interest, frame safety. Later sections provide the remaining details of our frame safety certifier.

Domain specific certification is different from programming language specific certification, such as memory safety or standard type checking. In programming language specific certification, such as in the PolySpace[15] tool, the abstract domain is a subpart of the programming language semantics. In domain-specific certification, the abstract domain is entirely separate; in fact our approach would allow it to be in a different logical system.

Conceptually, a domain-specific certifier consists of four main components, as shown in Figure 1. The programming language and the abstract domain are linked via symbolic evaluation and an abstraction function as described in the subsequent example. Domain-specific certification requires domain-specific annotations, which for the example in this paper require assertions on the program inputs. There are at least two ways to insert annotations for the inputs: one is to insert them at the beginning of the program and the other is to insert them where the inputs are used for the first time in the code. There are subtle trade-offs between these two options, involving the potential use of intermediate variables in expressions annotating input variables. The resolution of these trade-offs is beyond the scope of the paper; in our implemented system, program inputs are annotated at the beginning of the program.

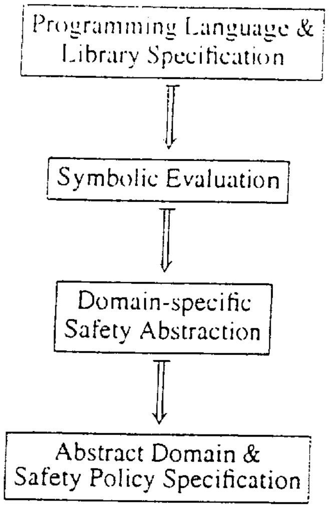

Figure 1. Domain-specific certifier.

2.1. Certifying Frame Safety

Coordinate frames underly all geometric calculations used in applications ranging from CAD to robotics to aerospace. They are implicit attributes of vectors and matrices, although they are not part of the computational domain. Geometric computations using vectors and matrices that have inconsistent coordinate frames yield meaningless results. Determining consistent use of coordinate frames normally requires detailed human analysis of a program. This section describes an implemented system which certifies the consistent use of coordinate frames for space observation geometry calculations.

Abstractly, a coordinate frame consists of an origin and an *orientation* which is specified by three orthogonal directions. With a coordinate frame, any point in space can be uniquely represented in rectangular coordinates by three real numbers. Directions can be represented by three real numbers, though not uniquely — the numbers representing the direction can be multiplied by any scale factor without changing the direction. Orientations are used abstractly to define directions and rotations. Rotations map an orientation onto another orientation. A *translation* maps one coordinate frame to another coordinate frame with the same orientation but a translated origin.

Our membership equational logic axiomatization of coordinate frames, described in Section 4.1, is restricted to the subdomain of frames that arises in the NAIF domain, namely those that can be inductively built out of predefined constant frames (such as `j2000`), ephemeris of planetary bodies, and the operations of rotation and translation. The `j2000` frame is a standard used for astronomical observations in the period 1975 to 2025. *Ephemeris* data specify the position and orientation of planetary bodies in our solar system relative to a constant frame. In the domain of space observation geometries, a typical example of a frame is one whose origin is at the center of a planet, whose Z axis lies along the north pole, and whose X and Y axes go through some fixed, designated points on the equator of the planet. Since the planet orbits around the sun and also rotates around its own axis, the sequence of frames generated by a planet over time can be quite complex. Library routines in NAIF provide this ephemeris data.

The inductive definition of frames corresponds to constant frames, calls to these ephemeris routines, and input variables to a program. The metric properties of the standard three dimensional group of rotations are not important for this abstract subdomain. Our axiomatization is sound for both our subdomain and the metric domain of frames and rotations; however, it is not complete for the latter.

2.2. Programming Language and NAIF Library

We consider below a generic assignment-based programming language, similar to Fortran and C (the NAIF library has alternate implementations in both languages). The language has sorts and subsorts, operations are typed by tuples of input sorts and an output sort, and expressions are built up by application of operations to subexpressions, variables, and constants. The only statement is the assignment.

Our simple generic programming language has just a few sorts, which are prefixed by `Pl` (for programming language) to distinguish them from the other sorts. The sorts are `PlString`, `PlInteger`, `PlReal`, `PlVector`, and `PlMatrix`, with `PlInteger` being a subsort of `PlReal`.

The subroutine library in our case study is the SPICE library from the Navigation and Ancillary Information Facility (NAIF) group at the NASA Jet Propulsion Laboratory. The NAIF group created this library to aid in solving geometric problems that arise in planetary exploration. The library contains subroutines for: coordinate system and time system conversion; matrix and vector operations; geometric operations (such as finding the outward-facing vector that is normal to an ellipsoid at a point on the ellipsoid's surface); and calculations that compute the travel time of light between two objects.

Vectors are implemented as triples of reals in NAIF, though for our purposes it is sufficient to treat them as an unstructured sort. The binary vector functions `vadd` and `vsub` are the usual vector addition and subtraction. Two additional vector functions are particularly important: `vdist` gives the distance between two points specified by the given vectors, and `vsep` gives the angle between the two directions specified by the two given vectors.

Note that at the programming language level there is no enforcement that two vectors are represented in the same coordinate system, or frame — this can't even be stated in the type system of the programming language. However, by interpreting the programming language variables and operations into the abstract domain described in Section 4.2, our certifier will find whether these operations are applied inconsistently.

Rotations are implemented by matrices at the programming language level. More precisely, rotation matrices have the property that they are invertible and their inverse is exactly their transpose. A vector can be multiplied (`mxv`) by a rotation matrix or by the transpose of a rotation matrix (`mtxv`), with the intuition that the frame in which the vector is defined is rotated.

Besides those above, the NAIF library functions used in this paper's example are the following: `utc2et` takes as input a time represented in "Universal Coordinated Time" calendar format (a string) and returns a time in the "Ephemeris" time system (a real number), which is an internal format used by other SPICE subroutines; `bodvar` takes as input the Id (an integer) of a solar system body and returns the radii of a solar-system body modeled as an ellipsoid; `georec` takes as input a point represented in geodetic coordinates and returns rectangular coordinates; `bodmat` takes as input a body Id and an ephemeris time, and returns a matrix describing the rotation of the body at that time relative to the standard `j2000` coordinate frame; `findp` takes as input a body Id and an ephemeris time, and returns the `j2000` position coordinates of a given body; `surfnm` takes as input 3 reals representing the radii of a body and a point on the body, and returns the outward vector that is normal to the surface at that point; and `sent` takes as input two body Ids and an ephemeris time, and computes the time a photon would have left one body so as to arrive at the other body at the time given to the subroutine.

The example programs which we have used in our case study come primarily from programs synthesized by Amphion/NAIF, including the representative one described below. The Amphion/NAIF synthesis system [11], given a high-level specification of a solar-system observation geometry problem, synthesizes a program, consisting of calls to SPICE subroutines, that solves the problem. As a test of our certifier, the programs generated by Amphion/NAIF were mutated by hand to yield many different unsafe programs. These were detected by our certifier. Also as a result of our case study, an axiom in Amphion/NAIF was found to be incorrect. The complementary roles of program synthesis and program certification are discussed in the concluding section of the paper.

An interesting lesson that we learned is that the full semantics of the target programming language is not needed for domain-specific certification. Domain-specific certification is not intended to be full program verification, rather, only specific aspects are certified. In our case study, we used a simple operational semantics of the programming language that enabled symbolic expressions to be calculated for the value of variables at each step of the program.

However, a domain-specific semantics is required for domain-specific certification. The symbolic expressions calculated from the operational semantics are lifted to the abstract domain level. The expressions at the programming level are calculated by a symbolic evaluation engine which, given a program and an expression containing variables defined in the program, calculates the canonical term associated to that expression that contains only functions and input variables.

2.3. Example Program

The example used in this paper is a representative program synthesized by the Amphion/NAIF system that calculates the angle at which Saturn appears at a given time from an observation point on the Earth's surface; the speed of light is also taken into consideration — Saturn appears to be in a slightly different place than it actually is because of the finite speed of light. The observation point is specified by geodetic (latitude, longitude, altitude) coordinates (`obsLLA`) and the time (`utcin`) is specified in Universal Coordinated Time calendar format (UTC). The angle is calculated between the outward normal to the Earth's surface at the observation point and the direction at which Saturn appears.

```
     et := utc2et(utcin) ;                    *** 1
 radear := bodvar(earthId, 'radii') ;         *** 2
   pobs := georec(obsLLA) ;                   *** 3
  dnorm := surfnm(radear, pobs) ;             *** 4
 mearth := bodmat(earthId, et) ;              *** 5
 tsatur := sent(saturnId, earthId, et) ;      *** 6
 dnorm2 := mtxv(mearth, dnorm) ;              *** 7
 pearth := findp(earthId, et) ;               *** 8
  pobs2 := mtxv(mearth, pobs) ;               *** 9
 psatur := findp(saturnId, tsatur) ;          *** 10
  pobs3 := vadd(pearth, pobs2) ;              *** 11
 dsatur := vsub(psatur, pobs3) ;              *** 12
   vang := vsep(dnorm2, dsatur) .             *** 13
```

The annotation on the input variable `obsLLA`, not shown here, says that, as a vector at the level of the programming language, it specifies abstractly a point on the Earth relative to the frame that is centered at the Earth at the input time `utcin` and rotated as the Earth is at that time. Indeed, this is because the geodetic position of that point was specified relative to the Earth as if the point was fixed to the Earth.

Line 1 converts the input UTC time into Ephemeris time system; line 2 calculates the three radii of the earth regarded as an ellipsoid; line 3 transforms the geodetic coordinates into rectangular coordinates; line 4 computes the normal to the Earth's surface at the given point, in the current Earth's coordinate frame; line 5 finds the rotation of the Earth at the given time relative to `j2000`; line 6 computes the time when light left Saturn so as to arrive at the Earth at the given input time; line 7 converts the normal at the specified point to a coordinate frame positioned as the current frame of the Earth, but having the orientation of `j2000`, and line 8 actually calculates the position of that frame in `j2000`; line 9 converts the given point on Earth into the same frame as the normal (see line 7); line 10 calculates the position of Saturn (also in `j2000`) at the time light left Saturn; line 11 converts the position of the given point to a position in `j2000` and then line 12 calculates the position of Saturn in the coordinate frame positioned at the given point and having the orientation of `j2000`; line 13 finally calculates the angle between the normal and the position of Saturn at the specified time.

The program above is relatively short but it is very easy to make domain-specific mistakes, especially if the code is written by hand, and these mistakes cannot be detected by common type checkers provided by programming languages. For example, one can forget line 7 which rotates the normal to the orientation of `j2000` and then calculates the angle between two directions in frames of different orientation. However, our certifier shows, in 1284 rewrites, that the program above is frame safe, while mutated versions are not. An example of the analysis performed by the certifier during the thousand-plus rewrites is that at line 11 it proves that the positions of the Earth and the given point on its surface, abstracted as translations, are composable; i.e., that their frames have the same orientation and the source frame of the point is the same as `j2000` translated to the position of the Earth. Section 4 describes the frame domain theory and abstraction in detail.

3. Technology

Membership equational logic and the language Maude are introduced in this section, providing the technical background for the formalization in the rest of the paper.

3.1. Membership Equational Logic

Membership equational logic [12, 1] is an extension of many-sorted equational logic [7] with membership assertions $t : s$ that state that a term t belongs to a sort s. It subsumes a wide variety of specification formalisms, including order-sorted [6, 8] and partial equational logics. Despite its generality, it still enjoys the good properties of equational logics: it is simple, efficiently implementable, and admits sound and complete deduction as well as free models. In this section we informally present membership equational logic, referring the reader to [12, 1, 2, 3] for a comprehensive exposition. We assume the reader is familiar with many-sorted equational logic.

In membership equational logic (MEL), the sorts are grouped in *kinds* and the operations are only defined on these kinds. A signature Ω consists of a set S of *sorts*, a set K of kinds, a map $\pi : S \rightarrow K$, and a $K^* \times K$-indexed set $\Sigma = \{\Sigma_{w,k} \mid (w,k) \in K^* \times K\}$ of *operations*. An Ω-algebra is a Σ-algebra A together with a subset $A_s \subseteq A_k$ for each $k \in K$ and each $s \in \pi^{-1}(k)$. For any K-indexed set of variables X, $T_\Sigma(X)$ denotes the usual (K, Σ)-algebra of terms. The sentences of MEL generalize the conditional equations $(\forall X)\ t \ = \ t'$ **if** C of equational logics by allowing membership assertions. These membership assertions are universally quantified Horn clauses of the form $(\forall X)\ t : s$ **if** C. In both types of sentences, the condition C is a finite set $\{u_1 = v_1, ..., u_n = v_n, t_1 : s_1, ..., t_m : s_m\}$ and $t, t', t_1, ...t_m, u_1, v_1, ..., u_n, v_n$ are terms in $T_\Sigma(X)$. If $n = m = 0$ then the sentence is *unconditional* or *atomic*.

This paragraph describes satisfaction of atomic sentence; the general case follows through the standard recursive definition. For an Ω-algebra A and an assignment $a : X \rightarrow A$, the function $a^* : T_\Sigma(X) \rightarrow A$ denotes the unique extension of a to a morphism of (K, Σ)-algebras. Thus A *satisfies* $(\forall X)\ t \ = \ t'$ (or $(\forall X)\ t \ : \ s$) if and only if for each assignment a, $a^*(t) = a^*(t')$ (or $a^*(t) \in A_s$). A MEL *specification* or *theory* is a pair (Ω, Γ), where Γ is a set of Ω-sentences, and it defines a class of Ω-algebras (those that satisfy it) denoted $\mathbf{Alg}_{(\Omega,\Gamma)}$.

The MEL proof theory is derived from the standard proof theory of equational logic. Its distinctive characteristic is that it allows the inference of the memberships of terms to sorts in addition to the standard equalities of terms. Given a specification (Ω, Γ), there are two rules that facilitate this inference. One rule is a modification of the *modus ponens* rule of equational logic to deduce a membership from a (conditional) sentence in Γ once its condition has been proven. The second rule is an extensionality rule over sorts which asserts that equal terms have the same sort.

$$\text{Membership:} \qquad \frac{\Gamma \vdash_\Omega (\forall X)\, t = t' \quad \Gamma \vdash_\Omega (\forall X)\, t : s}{\Gamma \vdash (\forall X)\, t' : s}$$

3.2. Maude

Maude [2, 3] is a high-performance rewrite system in the OBJ family [10] that supports membership equational logic. Its current version processes 800K rewrites per second on a 300MHz Pentium II. We use Maude notation in this paper to specify both the abstract domain knowledge and the programming language syntax, as well as the abstraction of the uninterpreted NAIF functions into the abstract domain. A few notational conventions are introduced next.

Equations and conditional equations are declared via the keywords `eq` and `ceq`, respectively; membership and conditional membership assertions are declared via the keywords `mb` and `cmb`. Operations can be declared using *mix-fix* notation, where underscores stand for arguments. They can also be overloaded; however, this is only syntactic sugar for appropriate conditional membership assertions. Declarations of the form `var X : S`, where `S` is a sort, are used to introduce variables; their scope is bounded by the enclosing *module*, introduced by `fmod ... end`. A module can import another module via one of the keywords `protecting`, `extending`, and `including`, or one of their shorthands `pr`, `ex`, and `inc`. The conditional membership assertions of MEL are denoted in Maude by `cmb X : S if C`; unconditional membership assertions by `mb X : S`. Subsort declarations have the form `S < S'` and are just syntactic sugar for a membership assertion: `cmb X : S' if X : S`. Kinds need not be declared explicitly. They are automatically calculated as the connected components of the partial order defined by subsort declarations, and one can refer to the kind of a sort `S` by using square brackets, `[S]`. The order of declarations is not important within a module.

A typical problem of specification formalisms that allow order-sorting and operator overloading is that some terms may have multiple correct sorts. The possible sorts of a term can be deduced using the complete deduction system of MEL, i.e., $sorts(t)$ is the set $\{s \in S \mid \Gamma \vdash_\Omega (\forall X)\, t : s\}$. A specification (Ω, Γ) is called *regular* if and only if for each term t, $sorts(t)$ is either empty or has a minimal element with respect to the subsort relation. A detailed discussion on regularity can be found in [1], together with decidability results and various syntactic criteria that imply regularity. Maude implements some of these criteria, and also warns the user when it cannot deduce regularity.

3.3. Abstraction, Partiality, and Safety in Maude

The least sort of a term in membership equational logic generalizes the standard notion, for finite lattices, of the least abstract type of an expression in the context of abstract interpretation. This generalization is defined in [5], where an environment for specifying and verifying abstract interpretations is presented. This paper extends the approach of this previous paper by considering partiality via least sort calculations. This section gives the reader the intuition through presentation of a simple example.

At the abstract level, let us consider a signature with two top sorts, one called `Frame` and another called `FrameSafe`, the second having various subsorts, including one called `Translation`. This signature has two kinds, and `[Translation]` and `[FrameSafe]` coincide. A translation can be thought of as taking a frame into another frame, so one can consider two operations, `sourceFrm` and `targetFrm`, of arity `Translation -> Frame`. A translation can always be inverted and translations can be composed, but not always: the target frame of the first translation must be the same as the source frame of the second. In Maude, one defines an operation `-_ : Translation -> Translation` and an operation `_+_ : Translation Translation -> [Translation]`. This second operation is partial: the result sort of a composition is a well-formed term of kind `[Translation]`, but more information is needed in order for it to become of sort `Translation`. The kind includes the output of any application of `_+_`, the sort includes just the output of well-defined applications. The sort can be inferred using a conditional membership assertion stating when translations are composable:

```
cmb Tsl + Tsl' : Translation
   if targetFrm(Tsl) = sourceFrm(Tsl') .
```

Unless other membership axioms for composition are given, the only way by which the result of a composition can be of sort `Translation` (and not just the more general kind `[Translation]`) is to actually prove the condition of the statement above. The detailed methodological approach to partiality in membership algebra is described in [12].

Once it is inferred that the least sort of a term is a subsort of `FrameSafe`, it means that that term represents a certifiably safe computation. In particular, if the sort of the composition of two translations is `Translation` then the two were safely composed. Notice that `FrameSafe` can have many distinct subsorts standing for various abstract types of entities in the abstract domain, such as orientations, rotations, directions, etc. All the sort inference computations are done at the abstract level.

The concrete expressions manipulated by the programming language are first abstracted via an operation

```
op |_| : Value -> [FrameSafe] .
```

which is defined recursively on the syntactic constructors of values that are manipulated by the programming language. Vectors and matrices are among these values, so they are defined as appropriate subsorts, i.e., `PlVector PlMatrix < Value`. For example, the subtraction of vectors, implemented by `vsub` in Fortran and similarly in our generic programming language, i.e., `vsub : PlVector PlVector -> PlVector`, is abstracted as `| vsub(V,V') |` of sort `[FrameSafe]`. The semantics of programming language functions is interpreted in the abstract domain, such as the following interpretation for `vsub`:

```
eq | vsub(V,V') | = (- | V' |) + | V | .
```

This interprets binary vector subtraction into the abstract domain as unary translation inverse on the first argument followed by composition of translations. The abstract type of `vsub` is inferred automatically. A somewhat different example, where the abstract type cannot be inferred automatically and additional attributes need to be provided, is the function `findp : BodyId Time -> Vector`. This is a NAIF library function that takes a body identifier, such as the earth's, and a time, and returns a vector representing the position of that body at the specified time in a standard coordinate frame, called `j2000`. We represent this abstract knowledge as follows:

```
mb | findp(B,T) | : Translation .
eq sourceFrm(| findp(B,T) |) = j2000 .
```

For this function the abstract type `Translation` is given explicitly and the source frame — a constant — is given.

Thus, the certification process can be viewed as: abstraction followed by least sort computation using abstract domain knowledge. If the least sort computation yields a safe sort for each abstracted expression calculated in the program, then the program is certified as safe for the domain-specific properties. For the frame-safety certifier described in this paper, all subsorts of `FrameSafe` are safe. In order to perform these sort inferences, abstract assertions about the inputs to a program are also needed. This information is given as annotations in the program.

4. Abstract Frame Domain Theory

In this section, we describe the abstract domain and present a figure describing its structure. Due to space limitations, the Maude axiomatizations are omitted, but the frame abstraction module is presented in Section 4.2.

4.1. Frame Domain

There are six sorts in our domain: `Real`, `Orientation`, `Rotation`, `Translation`, `Direction`, and `Frame`, and `Translation` is a subsort of `Direction`. This means that any translation defines a direction. Technically, any operation that can be applied to directions, such as the angle between two directions, can also be applied to translations; in this way, redundancy is reduced because some operations don't need to be declared four or more times to cover all combinations of translations and directions.

The Category of Rotations

Rotations are abstract objects whose function is to rotate other objects in the domain. These other objects are labeled with orientations or frames — and each frame has an orientation. Thus in our domain we can restrict the rotations to *labeled rotations* with an explicit source orientation and target orientation. Labeled rotations, together with orientations, form a partial algebra called a *category*. In terms of category theory, the objects of this category are orientations and the morphisms are rotations. Each rotation has an inverse formed by reversing the label of source and target. This axiomatization for labeled rotations is sound but not complete for the general group of unlabeled three dimensional rotations. Because it is sound, programs certified using this abstract domain are safe.

Each rotation has a source and a target orientation, given by the operations `sourceOrt` and `targetOrt`. There is a unit rotation for each orientation which behaves exactly like an identity morphism in a category, and an inverse for each rotation. The composition of rotations is a partial operation, requiring the target of the first rotation to match the source of the second rotation. The reason for this restriction to labeled rotations is that we don't want to allow programs that do meaningless and frame-unsafe calculations, such as to rotate a direction that is normal to the surface of the Earth at a given point by the rotation of the frame of a moon of Saturn relative to Saturn. Thus in order for a direction to be safely rotated, the certifier must prove that the orientation of the frame in which that direction is represented coincides with the source orientation of the rotation. As mentioned previously, the equations for labeled rotations are those of a category, as are the equations for frames and translations below.

Frames, Translations and Directions

Frames are formalized as an interrelated abstract data type to orientations. The orientation of a frame `Frm` is the term `frameOrt(Frm)`. A rotation can be applied to a frame if its source orientation is the orientation of the frame.

Translations also form a category, where the objects are the frames, and the morphisms are translations labelled by a source frame and a target frame. This part of the axiomatization is similar to those of rotations. Additionally, translations can be rotated. A common operation between translations with the same source frame is finding the distance between the origins of their target frames, denoted `[| tsl1, tsl2 |]`.

By abuse of language, the orientation of the source frame of a translation is often called the *orientation of the translation*. This is formally defined by the operation `frameOrt` (which thereby overloads the operation with the same name on frames). The orientations of the source and target frames of a translation are the same. Both the rotation of a translation and the distance between two translations are partial operations. For a translation `Tsl` to be rotatable by a rotation `Rot` (denoted by `@`), the orientation of the translation must be the same as the one of the rotation, while the distance between two translations makes sense if and only if the two translations have the same source frame. The orientation of a translation is also needed when one wants to treat a translation as a direction, for example, when one wants to calculate the angle between a translation and a direction, such as, the angle between the translation to Saturn and a direction normal to the surface of the Earth.

`Direction` is another abstract type. Directions can be thought of as the equivalence class determined by the relation of parallelism on the image of the function which forgets lengths of translations. We consider that a direction comes automatically with any translation. Unlike translations which carry a whole frame with them, directions only need to carry their orientation, that is, the orientation of the frame in which they were defined. Directions can also be rotated, if the source orientation of the rotation matches the orientation of the direction. A standard operation (denoted by `[< d1, d2 >]`) is finding the angle between two directions, which makes sense if and only if the two directions have the same orientation.

The abstract domain also contains an auxiliary useful frame constructor that was used in an annotation to specify the frame of an input variable, as mentioned in Section 2.3. The frame constructor builds a frame from a translation and a rotation relative to `j2000`.

The ADJ diagram [9] in Figure 2 depicts the sorts and the operations that form the abstract domain. The sorts are boxes and the operations are multisource arrows, each source standing for an argument. The operations having a circle index are partial. Because of space considerations we didn't draw the frame constructor explained above.

It is well known that many, if not most, domains of interest do not admit complete finite or not even recursively enumerable axiomatizations. Perhaps the most notorious example is the domain of natural numbers. When designing a domain specific certifier, one should aim toward soundness and clarity with respect to the domain rather than completeness. In this way, programs which are certifiably safe are indeed domain safe. After all, the purpose of certification is not to allow tricky and convoluted programs, but rather a reduced set of absolutely safe programs with respect to the intended policy. Therefore, the reader should regard our frame safety certifier as an experiment instantiating a general approach to domain specific certification.

4.2. Frame Abstraction and Safety Policy

Abstraction is the link between the programming language and the abstract domain. Each value computed by the program is first symbolically evaluated, then abstracted and checked for frame consistency at the abstract level. This last step is done in Maude via its least sort inference mechanism, by first declaring a common supersort, `FrameSafe` of all the sorts that abstract concrete values, and then checking if the abstraction of each value has the sort `FrameSafe`. The abstraction operation `|_| : Value -> [FrameSafe]` is defined that calculates the abstract sort of each value. If such a sort exists then we say that the value calculated by the program is *certifiably frame safe*:

```
fmod FRAME-SAFETY-ABSTRACTION is
  pr PROGRAMMING-LANGUAGE .
  pr ABSTRACT-DOMAIN .
  sort FrameSafe .
  subsorts Orientation Rotation
```

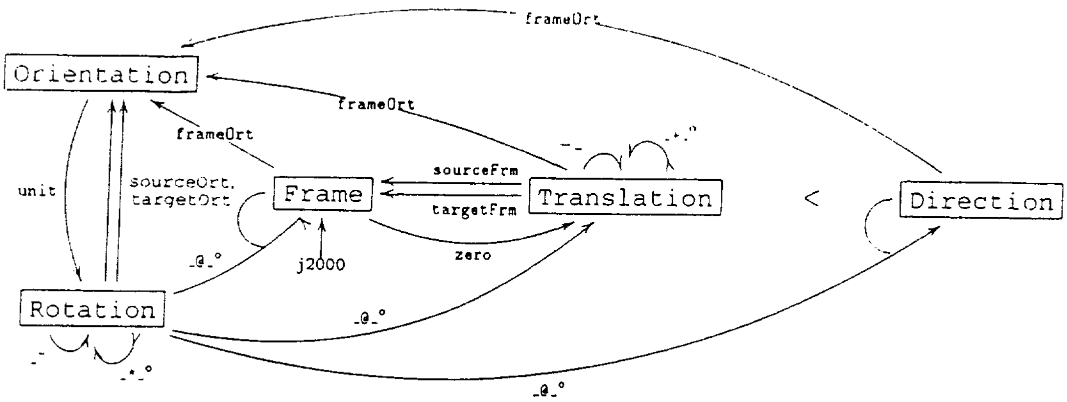

Figure 2. The ADJ diagram of the abstract domain of frames.

```
    Direction Translation Real < FrameSafe .

  op |_| : Value -> [FrameSafe] .

  vars B B' : BodyId .  var T : PlReal .
  var U : PlString .  var Rd : Attribute .
  vars R R' : PlReal .  var M : PlMatrix .
  vars V V' : PlVector .

  mb | utc2et(U)       | : Real .
  mb | bodmat(B,T)     | : Rotation .
  mb | findp(B,T)      | : Translation .
  mb | bodvar(B,Rd)    | : FrameSafe .
 cmb | sent(B,B',R)    | : Real if | R  | : Real .
 cmb | georec(V) | : Translation
    if | V  | : Translation .
 cmb | surfnm(V,V')    | : Direction
    if | V' | : Translation .
  eq | vdist(V,V') | = [| | V |,| V' | |] .
  eq | vsep(V,V')  | = [< | V |,| V' | >] .
  eq | vadd(V,V')  | = | V | + | V' | .
  eq | vsub(V,V')  | = (- | V' |) + | V | .
  eq | mxv(M,V)    | = | V | @ | M | .
  eq | mtxv(M,V)   | = | V | @ (| M | ~) .
  eq sourceOrt(| bodmat(B,T) |) = frameOrt(j2000) .
  eq sourceFrm(| findp(B,T)  |) = j2000 .
  eq sourceFrm(| georec(V) |) = sourceFrm(| V |) .
  eq frameOrt(| surfnm(V,V') |)
    = frameOrt(sourceFrm(| V' |)) .
endfm
```

The abstraction operation is defined recursively, in terms of abstract operations and sorts. The membership assertions above reflect the meaning of library functions in terms of abstract frame knowledge. In this section we ignore possible errors returned by library functions; these are addressed in the conclusion section on future work.

In the equation above, bodmat(B,T) returns a "safe" rotation matrix whose orientation is the frame orientation of j2000, and surfnm(V,V') is a direction whenever V' is a translation (the shape of the surface, V, doesn't affect the frame abstraction), its frame orientation being just the same as the orientation of the frame of the translation. Notice that there may be functions whose result is none of the sorts of interest for frame certification, such as bodvar which returns the three radii of a body modeled as an ellipsoid. In such situations, we just declare it FrameSafe; the least sort computation ensures that any use of such a value in a place where values with more concrete meaning are expected, such as translations or directions which happen to "look" the same at the level of programming language, will be reported as an unsafe use.

Finally, we define the frame safety policy as a predicate on programs. A program is safe if and only if each assigned value that is computed internally at each step of the program is frame safe. We call this strong requirement *stepwise safety*. A weaker requirement is *output safety*. For example, if a program written in an untyped language is supposed to return an integer value, say x, and it first executes x := n + 3.1 for a natural number n and then x := x + 2.9, then the program is output safe because the final computed value is an integer, but not stepwise safe. In our opinion, output safety is an insufficient requirement for certification.

As can be seen from the above description of the domain theory, sophisticated inferences for domain-specific certification can be carried out in membership equational logic. Furthermore, the axiomatization in Maude is compact and modular: the abstract domain consists of 36 axioms, the abstraction function consists of 24 axioms, and the frame policy consists of 2 axioms. Sort and subsort declarations add further semantic content, but the total Maude specification is only 250 lines.

5. Related Work

Certification technology based on static analysis is maturing to the point of commercial viability. PolySpace is such a tool that detects statically, via abstract interpretation techniques, errors that would normally occur at runtime, such as arithmetic exception (e.g., division by zero), illegal pointer dereferencing, overflow, read access to uninitialized data, out-of-bounds array access, etc. It is completely automatic in the sense that it works directly on the source code; no annotations are needed and no axiomatization of abstract domains required. However, it only detects what we call "programming language specific errors", being unable to reason about high level, domain-specific safety policies.

Extended Static Checker (ESC) [4, 14] is a tool that finds programming errors at compile time, such as array index bounds errors, nil dereferences, deadlocks and race conditions. The user of ESC annotates the programs with specifications in a precondition-postcondition style which are checked statically using a theorem prover for untyped predicate calculus with equality. The type system of the target programming language, Modula-3, is implemented in untyped first-order logic. The use of ESC is therefore limited to programming language definable types. In contrast, our approach to domain-specific certification totally separates the abstract domain of interest from the programming language. The abstract domain can be axiomatized using first-order many-sorted membership and rewriting logic and can be reused for various target languages. The user of the system can modify the abstract domain in a flexible manner.

6. Conclusion and Future Work

This paper described a general architecture for domain-specific software certification based on abstract interpretation. Similarities and differences with certification of policies at the level of programming language semantics were discussed. A case study of a significant domain-specific safety policy of interest to NASA, namely coordinate frame safety, was completed with an implementation of the general architecture in Maude. We note that the modules for the abstract domain are independent of the programming language level, and hence could be reused in different contexts. The implementation in Maude will facilitate experimentation with other domain-specific safety policies.

The case study was done in a domain for which we had previously developed a program synthesis system. The motivation was to investigate the relationship between program synthesis and program certification. This investigation is still underway, and is an extension of our work in automatically generating documentation for synthesized programs. In essence, documentation can be used by human reviewers in the manual certification of programs, while annotations can be used for the automated certification of programs. The annotations required in our current investigation on coordinate frame-safety can be restricted to the inputs of a program; the logical engine is sufficiently powerful to infer intermediate assumptions. In part of our future work we expect to investigate domain-specific safety policies where logical inference of intermediate assumptions is no longer tractable for a simple and trusted certification system, but checking of annotations provided by program synthesis is tractable. Although our synthesis system is based on deductive technology, the addition of extensive decision procedures to enable tractable program generation raises additional burdens in showing the correctness of the generated programs. This burden can be met either by certifying the correctness of the entire program synthesis system or individually certifying the generated programs.

The abstract domain of the certifier was developed independently from the domain theory of the program synthesis system. It is related to a subset of the Amphion domain theory, but is formulated quite differently since it is oriented towards bottom-up checking rather than top-down synthesis. Because it was developed independently, it provides an independent check for synthesized programs as well as for hand-written programs. In fact, a subtle error in the Amphion/NAIF domain theory was discovered: an extraneous matrix transpose operation was generated which caused the resulting synthesized programs to be frame unsafe.

6.1. Scaling Issues

Empirically, programs in the NAIF domain are usually no larger than one hundred lines of code (because of the substantial functionality of the NAIF component library). For programs in this range, the certification is so fast that the timing profile registers 0 milliseconds. We generated a suite of synthetic programs up to 1,000 SLOC to determine scaling properties, under two conditions: no caching of rewrite results, and limited caching of rewrite results (specifically, caching for the substitution and abstraction operators). At 1,000 SLOC, certification with no caching required 2.2 seconds, while certification with caching required 1.3 seconds. Without caching, it is expected that the certification scales quadratically, because each line requires computing a symbolic evaluation including all preceding lines. We validated this quadratic scaling with our suite of synthetic programs.

With caching, the symbolic evaluation does not need to be redone for each previous line. In principle, with optimized data structures, caching imposes close to linear overhead. However, the current implementation of Maude is not yet optimized for caching, and is known to introduce quadratic factors. Accordingly, our experiments of certification with caching also revealed a quadratic scaling of execution time in SLOC. While we believe that our approach to domain-specific certification potentially scales linearly with more finely optimized rewrite algorithms and data structures, even with a more conservative quadratic scaling extrapolation, programs in the range of 10,000 SLOC would only require minutes to certify.

6.2. Domain-specific Error Handling Certification

The NAIF library has a mode that allows the programmer to handle errors that are signalled in NAIF library routines. In this mode, a NAIF subroutine simply returns if it detects an internal error. Subsequent to calling the NAIF subroutine, the boolean function `failed()` can be called to find out whether an error was detected, and the subroutine `getsms` can then be called to find out which error occurred. In particular, the NAIF routine `bodmat` detects several errors, one of which occurs when the data necessary to compute the rotation matrix for the given body and time has not been loaded.

We are currently extending our domain-specific certifier to check statically whether *all* possible errors due to the NAIF subroutines are properly handled. There are three situations that should be considered when a NAIF subroutine is called: 1) it executes normally without errors; 2) it flags an error and the returned value is subsequently used in the program; 3) it flags an error but the returned value is not used. Only the second situation is unsafe, so the certifier must detect it.

The least-sort computation of Maude is very suitable to implement such a domain specific error handling certifier: we declare the abstractions of all library functions that can return errors to be of target `[FrameSafe]`, i.e., they are seen as partial functions, and then write conditional membership assertions and/or equations that state when such a function actually returns a proper value and/or when it returns an error and of what kind. Thus, if the sort of the abstraction of a value calculated by the program turns out to be a subsort of `FrameSafe`, then it means that all the conditions of the associated membership assertions were proved, that is, all the errors that can possibly be generated by the function that returned the value will either provably not occur or they were already handled by the program through conditional statements guarded by the function `failed()`.

References

[1] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[2] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic, March 1999. Maude System documentation at `maude.csl.sri.com/papers`.

[3] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. The Maude system. In Paliath Narendran and Michaël Rusinowitch, editors, *Proceedings of RTA'99*, volume 1631 of *LNCS*, pages 240–243. Springer-Verlag, July 1999.

[4] Compaq. Extended Static Checking for Java, 2000. URL: `www.research.compaq.com/SRC/esc`.

[5] Bernd Fischer and Grigore Roşu. Interpreting abstract interpretations in membership equational logic. Technical Report TR 01-16, RIACS, May 2001.

[6] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[7] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

[8] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[9] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. A junction between computer science and category theory, I: Basic concepts and examples (part 1). Technical report, IBM Watson Research Center, Yorktown Heights NY, 1973. Report RC 4526.

[10] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[11] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proc. 9th Knowledge-Based Software Engineering Conference*, pages 48–57, 1994.

[12] José Meseguer. Membership algebra as a logical framework for equational specification. In *Proceedings, WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[13] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[14] K. Rustan, M. Leino, and Greg Nelson. An extended static checker for modula-3. In K. Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, April 1998.

[15] PolySpace Tech. URL: `www.polyspace.com`.