Certifiable specification and verification of C programs

Christoph Lüth and Dennis Walter

Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany
Christoph.Lueth@dfki.de, Dennis.Walter@dfki.de

Abstract. A novel approach to the specification and verification of C programs through an annotation language that is a mixture between JML and the language of Isabelle/HOL is proposed. This yields three benefits: specifications are concise and close to the underlying mathematical model; existing Isabelle theories can be reused; and the leap of faith from specification language to encoding in a logic is small. This is of particular relevance for software certification, and verification in application areas such as robotics.

1 Introduction

Software verification is used to many ends, and each end has its own means. In this paper, we present an approach for the specification and verification of mathematically-oriented C programs in the context of software certification, where correctness and reliability of the verification process are a major concern. Therefore we emphasise trustworthiness and correctness, by reduction to a proof in a trusted theorem prover (in our case, Isabelle/HOL), such that the only leap of faith required is the embedding of the semantics of the programming language into the prover, and the actual definition of correctness.

To ensure consistency between specification and program code, we annotate the source code with specifications of its intended behaviour, but instead of extending the programming language with specification constructs (as in JML or ACSL [1, 2]), we use the higher-order logic of the underlying prover, extended by convenient ways to relate to values of the program state. In our application domain, safety software in robotics[1], programs live in a fairly rich application domain, involving in particular concepts from geometry such as points, lines, or convex polygons. Using the prover's native higher-order language, specifications become concise and easy to read, since these concepts lend themselves well to higher-order formalisation. Moreover, we can use Isabelle's rich libraries.

The actual correctness proofs follow previous work [3–5]: we encode the programming language and its semantics into the theorem prover, define a semantic way of when a specification is satisfied, and show the verification rules as theorems. Verification conditions are generated (and proven) in Isabelle. However,

[1] http://www.sams-project.org/

$BaseLoc_0$	$Type_0$	$Val_{0,0}$	$Val_{0,1}$	$Val_{0,2}$	$\cdots$
$BaseLoc_1$	$Type_1$	$Val_{1,0}$			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$BaseLoc_n$	$Type_n$	$Val_{n,0}$	$Val_{n,1}$		

Fig. 1. A state maps base locations to types and sequences of scalar values

some work is necessary to scale this up for realistic programs: one has to make sure the proof state remains manageable, and we have to provide a modular way to verify each function separately. Hence, our main contribution is an approach which allows comprehensible, concise specifications of C programs in a rich problem domain, together with a well-defined, rational verification process, by tight integration of programming language and theorem prover.

The paper is structured as follows: Sec. 2 describes the semantic foundations, such as the supported C language subset, its denotational semantics, the semantic notion of specification, satisfiability, and how it is proven. Sec. 3 introduces the novel hybrid specification language we annotate program functions with, followed by an account on how programs are formally verified, given in Sec. 4. We look at related work and conclude in Sec. 5.

2 Semantic Foundations

Overall, our model is a text-book semantics as found in e.g. [6]. Its distinctive features are a deep embedding of a (subset of) the C language, to which we give a deterministic denotational semantics as a mapping from states to states, and a shallow embedding of functions.

2.1 The Low-level State Model

The *state* represents a program's memory. Abstractly, it is a map from locations *Loc* to values *Val*. In contrast to the usual memory model as a stack of local variables and a heap containing allocated objects, we use a flat model where all objects are given a *base location*. An *object* is defined in the C standard as a "region of storage [...], the contents of which can represent values" [7, 3.14]. We represent it as a sequences of *scalar* values [7], modelled as partial functions from $\mathbb{N}$ to *Val*. Scalar values are integer and floating-point numbers—which we model as unbounded integers and reals—and references, $Val = Int + Real + Ref$, where a reference is a location or undefined (the null pointer) $Ref = Loc + 1$. Fig. 1 depicts the structure of the state space.

Concretely, a state is then a finite partial function $\Sigma : BaseLoc \rightharpoonup (Type \times (\mathbb{N} \rightharpoonup Val))$ mapping base locations to representations of objects and their (run-time) type *Type*. To access scalar values (possibly inside structures or arrays) we use *locations*, which are pairs $Loc = BaseLoc \times \mathbb{N}$. Thus, locations represent addresses. They allow a limited form of arithmetic, as defined in the standard:

Supported	Not supported
Addresses of local objects (via &)	Casts to and from **void** $*$
Pointer offsets and subtraction	Function pointers
sizeof operator	**switch**, **goto**, **continue**
Function calls in expressions	Arbitrary side effects

Fig. 2. Summary of supported and unsupported language features

we can add and subtract the offsets of locations sharing the same base location. (A simplifying assumption here is that all scalar values have the same size.) Our model precludes the use of structured values in expressions, so they cannot occur as arguments to assignment or functions. The basic operations on states are reading, writing, allocation and deallocation:

$$read : Loc \to \Sigma \to Val \quad update : Loc \to Val \to \Sigma \to \Sigma$$
$$\mathit{fresh\text{-}loc} : \Sigma \to BaseLoc \quad extend, dealloc : BaseLoc \to \Sigma \to \Sigma$$

The *fresh-loc* operator returns a base location that is not yet used. Deallocation is currently used for local variables on function exit; *malloc* and *free* could easily be supported by our model as well. State updates always succeed, i.e. at the state level we do not perform type checks, array bounds checks or pointer validity checks. Sanity checks are instead inserted into the semantics of pointer dereferencing and array access.

We do not follow the split heap approach [8] literally, but recover the needed inequalities through appropriate lemmas on field and array access. This keeps the state model reasonably close to the C standard.

2.2 Modelled Language Subset

We support a subset of the language given by the MISRA programming guidelines [9] (Fig. 2). Prominent features include those which are heavily used in our application domain: arbitrary nesting of structures and arrays, limited form of address arithmetic, function calls in expressions, and an &-operator that can be used on both global and local objects. Fig. 3 shows excerpts of the datatypes modelling the language. An external front-end parses the actual C source code into the Isabelle datatypes, and also performs static checks for type correctness, and conformance to our language subset, including the MISRA guidelines.

A rather drastic simplification from a theoretical point of view is the exclusion of recursive functions by the guidelines [9, Rule 16.2]. This allows us to give semantics to functions without the need for an explicit fixed-point operator.

Since expressions can have side-effects, evaluation order would be important. However, [9, Rule 12.2] requires that an expression must yield the same value under every evaluation order. We check this on the syntactic level in the front end, by ruling out problematic expressions such as `(x= 2)/x−−`, but allowing calls to side-effect free functions. As we only consider MISRA-conformant programs, it is adequate to fix the evaluation order, proceeding from left to right for both function argument lists and expression trees.

$$
\begin{array}{rcl}
\textit{basic-type} & ::= & \textbf{int} \mid \textbf{double} \mid \textbf{void} \\
\textit{type} & ::= & \textit{basic-type} \\
 & \mid & * \ \textit{type} \\
 & \mid & \textbf{struct} \ \textit{id} \ (\textit{id} \times \textit{type}) \ \textit{list} \\
 & \mid & \textit{type} \ [\textit{Nat}] \\
 & & \\
\textit{stmt} & ::= & \textbf{while} \ \textit{expr} \ \{ * \ \textit{stmt} \} \\
 & \mid & \textit{lval} = \textit{expr} \\
 & \mid & \textit{id} \, (\textit{expr list})_{\mathsf{stmt}} \\
 & \mid & \textit{stmt}; \ \textit{stmt} \mid \ldots \\
 & & \\
\textit{id} & ::= & \textit{String} \\
\textit{arith-op} & ::= & + \mid - \mid * \mid \ldots \\
\textit{comp-op} & ::= & == \mid != \mid < \mid \ldots
\end{array}
\qquad
\begin{array}{rcl}
\textit{lval} & ::= & \textit{id}_{\textit{type}} \\
 & \mid & \textit{ref-expr}[\textit{int-expr}]_{\ \textit{type}} \\
 & \mid & \textit{lval}.\textit{id}_{\ \textit{type}} \\
 & \mid & * \ \textit{ref-expr}_{\ \textit{type}} \\
\textit{ref-expr} & ::= & \textbf{NULL} \\
 & \mid & \textit{lval} \\
 & \mid & \& \ \textit{lval} \\
 & \mid & \textit{id} \, (\textit{expr list})_{\mathsf{ref}} \\
\textit{int-expr} & = & \textit{Int} \\
 & \mid & \textit{lval} \\
 & \mid & \textit{int-expr comp-op int-expr} \\
 & \mid & \textit{int-expr arith-op int-expr} \\
 & \mid & \textit{id} \, (\textit{expr list})_{\mathsf{int}} \mid \ldots \\
\textit{expr} & = & \textit{int-expr} \mid \textit{double-expr} \mid \textit{ref-expr}
\end{array}
$$

Fig. 3. The datatypes of the deep C embedding (abridged). Names of datatypes are denoted *in italics*, and constructors written in concrete syntax. Note how we annotate lvalues with their type, such that we can compute necessary state offsets easily.

2.3 Denotational Semantics

Our semantics is deterministic and identifies all kinds of faults like invalid memory access, non-termination, or division by zero as complete failure. We use the overloaded semantic brackets $[\![-]\!]$ for all semantic functions, which assign a meaning to each of the datatypes modelling programs, amongst them

$$[\![\mathit{stmt}]\!] : \Gamma \to \Sigma \rightharpoonup 1 \times \Sigma \qquad [\![\mathit{expr}]\!] : \Gamma \to \Sigma \rightharpoonup \mathit{Val} \times \Sigma \qquad [\![\mathit{lval}]\!] : \Gamma \to \Sigma \rightharpoonup \mathit{Ref} \times \Sigma$$

where Γ is an environment which maps identifiers to locations. Note that in the presence of pointers, the evaluation of an lvalue (e.g. $*\mathsf{x}$) depends on the state. State transformers are composed with the Kleisli composition [10] (where α and β are type variables) $\gg= : (\Sigma \rightharpoonup \alpha \times \Sigma) \to (\alpha \to \Sigma \rightharpoonup \beta \times \Sigma) \to \Sigma \rightharpoonup \beta \times \Sigma$ which passes the result tuple of the first argument into the second function. The semantic function for an assignment evaluates the lvalue l to a location m, then the rvalue e side to a value v, and uses *update* to change the state:

$$[\![l = e]\!]\Gamma \stackrel{def}{=} [\![l]\!]\Gamma \gg= \lambda m.\ [\![e]\!]\Gamma \gg= \lambda v.\ \mathit{update}\ m\ v$$

The conditional statement and iteration are interpreted by corresponding operations on state transformers. A bounded iteration operator models the unfolding of the loop at most n times, and the semantics of iteration is the least number of unfoldings to make the loop condition false, if it exists, and undefined otherwise.

We consider the idealisation of machine integers to mathematical integers a sensible separation of concerns, as the absence of under-/overflow can in many practical cases be proven by means of abstract interpretation [11], and using

modular arithmetic makes interactive verification unbearably cumbersome. Modelling floating-point numbers as real numbers ignores the issue of numerical precision. For the time being, we simply do not treat it formally.

2.4 Modelling Functions

Functions are modelled as HOL functions. The semantic function of a C function with n parameters takes n values, and returns a state transformer:

$$[\![\mathsf{type}\ \mathsf{id}(x_1, \ldots, x_n)\ \mathsf{block}]\!] : \Gamma \to Val^n \to \Sigma \rightharpoonup Val \times \Sigma$$

Both parameters and local declarations are translated into state extensions, and the new locations added to the environment, but parameters are initialised with the argument values, and are visible in the pre- and postconditions of the function. Ignoring specifications — which are also included in the full environment — the environment Γ maps variables to their allocated base location, and function identifiers to their semantics:

$$\Gamma \cong (Id \rightharpoonup BaseLoc) \times (Id \rightharpoonup (Val^n \to \Sigma \rightharpoonup Val \times \Sigma))$$

When calling a function f, we evaluate the n argument values, look up its value in the environment, written as $\Gamma\ !\ f$, and call the resulting state transformer:

$$[\![\mathsf{f}(\mathsf{args})]\!]\Gamma \stackrel{def}{=} [\![\mathsf{args}]\!]\Gamma \gg\!= (\Gamma\ !\ \mathsf{f})$$

2.5 Specifications

Semantically, we consider specifications to be state predicates. In the classic total Hoare calculus, a specification for a program p consists of a precondition P and a postcondition Q, written $[P]p[Q]$. In our typed setting, the precondition is a state predicate $P : \Sigma \to bool$, the program is a state transformer $p : \Sigma \rightharpoonup \alpha \times \Sigma$, and the postcondition a predicate over the state and the result of the program, $Q : \alpha \times \Sigma \to bool$. The specification is satisfied by p if each state satisfying P is mapped to one satisfying Q:

$$\begin{aligned} &\models : (\Sigma \to bool) \to (\Sigma \rightharpoonup \alpha \times \Sigma) \to (\alpha \times \Sigma \to bool) \to bool \\ &\models [P]\, p\, [Q] \stackrel{def}{=} \forall S.\ P\ S \longrightarrow def(p\ S) \wedge Q(p\ S) \end{aligned}$$

To show that a program satisfies a specification, we introduce a *syntactic* notion of satisfiability for each datatype in Fig. 3, which is defined in terms of the semantic notion (shown here for expressions and statements)

$$\begin{aligned} &\vdash_e : \Gamma \to (\Sigma \to bool) \to expr \to (Val \times \Sigma \to bool) \to bool \\ &\vdash_s : \Gamma \to (\Sigma \to bool) \to stmt \to (1 \times \Sigma \to bool) \to bool \\ &\Gamma \vdash_e [P]\ e\ [Q] \stackrel{def}{=} \models [P]\, [\![e]\!]\Gamma\, [Q] \qquad \Gamma \vdash_s [P]\ s\ [Q] \stackrel{def}{=} \models [P]\, [\![s]\!]\Gamma\, [Q] \end{aligned}$$

2.6 Modular Verification

In order to handle realistic programs, verification needs to be modular, i.e. we want to verify each function in the program separately and use only its specification during the verification of its callers. Further, we want to keep the specification of each function *local* to its direct effects. For simple imperative languages, this is achieved by *frame rules* [12]. The presence of pointers complicates the situation as possible aliasing forces these rules to become inelegant and complex. Our solution is to make changes to the state possibly caused by a function (semantically, a state transformer) part of the specification. Recall that the state is essentially a finite map. We restrict a state S to a set of locations L by a restriction operation $S\restriction_L$, and define the *modifies* predicate for two states S, T and a set of locations Λ which holds if S and T agree everywhere except for Λ:

$$S \simeq_\Lambda T \stackrel{def}{=} S\restriction_{\neg\Lambda} = T\restriction_{\neg\Lambda}\ .$$

We extend the notion of satisfaction by a *modification set*, which contains the only locations this state transformer may change, thus effectively integrating the frame rule into the notion of satisfaction.

$$\begin{aligned}
&\models : \mathit{Loc\ set} \to (\Sigma \to \mathit{bool}) \to (\Sigma \rightharpoonup \alpha \times \Sigma) \to (\alpha \times \Sigma \to \mathit{bool}) \to \mathit{bool}\\
&\Lambda \models [P]\, p\, [Q] \stackrel{def}{=} \forall S.\ P\ S \longrightarrow \mathit{def}(p\ S) \wedge Q(p\ S) \wedge S \simeq_\Lambda (p\ S)\\
&\vdash_s : \mathit{Loc\ set} \to \Gamma \to (\Sigma \to \mathit{bool}) \to \mathit{stmt} \to (1 \times \Sigma \to \mathit{bool}) \to \mathit{bool}\\
&\Lambda, \Gamma \vdash_s [P]\ s\ [Q] \stackrel{def}{=} \Lambda \models [P]\, [\![s]\!]\Gamma\, [Q]
\end{aligned}$$

The syntactic proof rules integrate checks that only locations in the modification set are modified. Sec. 4.2 gives further details, and an example (Fig. 7).

3 Program Specifications with Isabelle

Programs are specified through *annotations* embedded in the source code in specially marked comments (beginning with `/*@`, as in JML or ACSL). This way, annotated programs can be processed by any compiler without modifications. Annotations can occur before function declarations, where they take the form of *function specifications*, and inside functions in front of loops, where they serve as *loop specifications*, which play a technical rôle in that they allow automatic generation of verification conditions. A function specification consists of a precondition (`@requires`), a postcondition (`@ensures`) which relates the state before entry into the function (the *pre-state*) to the state after the function has returned (*post-state*), and a modification set (`@modifies`). Loop specifications consist of an invariant (`@invariant`), a variant (`@variant`; a measure function on program states, mapping program states to $\mathbb{N}$ given an appropriate C expression) ensuring termination of the loop, and an optional modification set; Fig. 4 gives an example. We first explore the design rationale before delving into the technicalities.

```
/*@ @modifies a[0:len], *res @*/
void avg(int *a, int len, double *res) {
  int i;
  /*@ @modifies i, *res, a[0:len]
      @variant len - i            @*/
  for (i=0; i<len; ++i) { *res += a[i]; a[i] = 0; }
  *res /= len;
  }
```

Fig. 4. Annotated function demonstrating features as found in e.g. JML.

3.1 Design Rationale

Our aim is to specify and verify program modules for the domain of safety-relevant robotics and automation. Functions in these programs often represent mathematical operations. They range from simple vector operations (scalar product, transformations) over computing the convex hull of a point set to the approximation of the behaviour of a moving object. The data structures these operations work upon is rather restricted. For simplicity and memory safety, they tend to be static; dynamic objects are sparse. In many cases, these data types are structurally just tuples and sequences of real numbers and integers.

In contrast to their representations, the objects of interest in the mathematical domain are not necessarily discrete and finite. They include time-dependent functions, areas, polygons etc. Therefore, to obtain the desired degree of abstraction and detail in function specifications, an expressive, mathematically oriented language is required. Fundamental concepts like real numbers, sets, geometric transformations, but also concepts from analysis like derivations, integrals or limits should be easily definable or preferably predefined. Moreover, for the actual verification, a plethora of lemmas about these operations and their interaction will be needed. Also, for readability, the language should be syntactically flexible (e.g. support infix notation), and have a larger glyph set than 7-bit ASCII.

Finally, we do not expect to be able to prove the domain-related parts of program specifications automatically, by calling provers like Z3 [13] or CVC3 [14], despite the impressive advances these tools have made. For the interactive proof work it is then a real benefit if only one formal language needs to be understood.

These considerations led to the decision to directly use Isabelle as the specification language for state predicates as used in pre-/postconditions and invariants, in contrast to JML and ACSL, where extensions of the programming language are used in specifications. However, we need a way to refer to the program state, and in particular the value of variables in the specifications, and further there are technical specifications like ranges of array indices or validity of pointers, that are best written down in the C syntax. This resulted in a hybrid approach, where Isabelle and an extension of the C syntax can be combined by a quote/antiquote-mechanism, combining the best of both worlds.

3.2 The Specification language

State predicates are boolean expressions over atomic formulas, formed by the operators &&, || , !, --> (implication) and <-> (equivalence), and the quantifiers \ **forall** T i; P and **exists** T i; P. An atomic formula is one of the following: *(i)* a side-effect free C expression of integer type (with a valuation of 0 denoting *false* and anything else *true*), which may additionally contain bound variables introduced by the quantifiers above, the operator **old**(e) referring to the value of expression e in the pre-state, and the special symbol **result** which refers to the function's return value; *(ii)* a pointer predicate; or *(iii)* a quotation.

Pointer predicates use keywords to state common properties of pointers. These are **valid**(p) (expressing validity of a pointer), **array**(a, n) (array a has at least n elements), and **separated**(a, m, b, n) (the memory areas denoted by a[0:m] and b[0:n] are fully disjoint arrays).

Quotations are the means to embed Isabelle terms of type *bool* into specifications, e.g. to formulate the domain-related parts of a specification. A quotation consists of an arbitrary Isabelle term enclosed in ${...}. Reference to the program state within quoted Isabelle terms is made possible via *anti-quotations*, which allow expressions in C syntax to be spliced into a quotation. Anti-quotations are syntactically enclosed in ‘{...}. Intuitively, an anti-quoted C expression is interpreted by its semantic value (Sec. 2.3). As an example, using the predefined Isabelle functions *cmod* and *Complex*, ${cmod (Complex ‘{x+1} ‘{y}) < c} expresses that the complex number $(x+1)+iy$ has a modulus below c, where x and y are program variables of floating-point type. As a shorthand, for identifiers one may write ‘x for ‘{x}, and $x for ${x}.

3.3 Translation to Isabelle

In contrast to programs, specifications are embedded shallowly as Isabelle functions, where the translation from the specification language to Isabelle is performed by the front-end. Preconditions P, postconditions Q and invariants I are translated to Isabelle functions of types

$$P : \Gamma \to \Sigma \to \mathit{Val}^n \to \mathit{bool} \qquad I : \Gamma \to \Sigma \to \mathit{bool}$$
$$Q : \Gamma \to (\Sigma \times \mathit{Val} \times \Sigma) \to \mathit{Val}^n \to \mathit{bool}$$

Note that the denotation of a modification set can also depend on the pre-state, e.g. to specify that ∗x is changed when passing a pointer x to a function. Modification sets are given the semantics $[\![\mathsf{mlist}]\!] : \Gamma \to \Sigma \to \mathit{Loc\ set}$, and the front-end simply outputs the modification set as a value of the datatype *mlist*.

The translation is performed by a collection of operations $\{\#, \#_l, \#_r, \#_i, \#_d\}$ on the abstract syntax of specification terms, for predicates, locations and the expression types. We only sketch the translation of preconditions, as the others are analogous. The generated Isabelle term for a precondition *Pre* will have form

$$\lambda\ \Gamma\ \Sigma\ (v_1, \ldots, v_n) \bullet \#(\mathit{Pre}) \tag{1}$$

$$
\begin{array}{lrcl}
\textit{Predicates:} & \#(\mathsf{A}\ \&\&\ \mathsf{B}) & \overset{def}{=} & (\#(\mathsf{A}) \wedge \#(\mathsf{B})) \\
 & \#(\backslash\textbf{valid}(\mathsf{p})) & \overset{def}{=} & (valid_ptr\ \Sigma\ ([\![\mathsf{p}]\!]\ \Gamma\ \Sigma)) \\
 & \#(\backslash\textbf{array}(\mathsf{p},\ \mathsf{n})) & \overset{def}{=} & (valid_arr\ \Sigma\ ([\![\mathsf{p}]\!]\ \Gamma\ \Sigma)\ ([\![\mathsf{n}]\!]\ \Gamma\ \Sigma)) \\
 & \#(\mathsf{E1} < \mathsf{E2}) & \overset{def}{=} & \#_x(\mathsf{E1}) < \#_x(\mathsf{E2}) \quad (x \in \{l, d\}) \\
 & \#(\$\{s_1\ `\{\mathsf{a1}\}\ s_2\ `\{\mathsf{a2}\} \cdots\ s_k\ \}) & \overset{def}{=} & (s_1\ \#_{x_1}(\mathsf{a1})\ s_2\ \#_{x_2}(\mathsf{a2}) \cdots s_k) \\
 & & & \qquad (x_i \in \{l, r, i, d\}) \\
\textit{Expressions:} & \#_x(\mathsf{E1} + \mathsf{E2}) & \overset{def}{=} & \#_x(\mathsf{E1}) + \#_x(\mathsf{E2}) \quad (x \in \{l, d\}) \\
 & \#_i(\mathsf{lval}) & \overset{def}{=} & int\,(read\ \#_l(\mathsf{lval})\ \Sigma) \\
 & \#_l(\mathsf{lval}[\mathsf{e}]) & \overset{def}{=} & array\text{-}acc\ \#_l(\mathsf{lval})\ \#_i(\mathsf{e}) \\
 & \#_i(\$\mathsf{a}) & \overset{def}{=} & a
\end{array}
$$

Fig. 5. Rules for the translation from abstract to Isabelle syntax

The translation operation # generates the *body* of the lambda term (1). This means we translate state predicates in the implicit context of an environment Γ, the pre-state Σ and function argument values v_i. We cannot define this translation within Isabelle in terms of the semantic functions $[\![\cdot]\!]$, since Isabelle code may appear in quotations, and antiquotations may refer back to bound variables introduced in quotations, but the translation via # resembles the defined expression semantics on the quotation-free part of a specification term.

Fig. 5 shows representative translation rules. The logical connectives are directly translated to their Isabelle equivalents. For each pointer predicate there is an Isabelle counterpart with the additional arguments Γ and Σ; e.g. *valid_ptr* interprets **valid**. Since the arguments to pointer predicates are unextended C expressions, we can use the semantic function to interpret the abstract syntax in Isabelle. Quotations are output verbatim, except for anti-quotations, the translation of which is spliced into the quotation on the point of occurrence. The translation of expressions behaves like the respective semantic function, but outputs references to bound Isabelle variables ($a) by their name ($a$).

3.4 Representation Functions

A *representation function* maps objects represented implicitly in the state to their explicit denotation. For example, in the specification stating that the sum of the elements of a vector p.v is less than δ: ${ $sum\ (Vec\ \Sigma$ `{p.v} `{p.vlen}$) < \delta$} the function $Vec :: \Sigma \rightarrow Loc \rightarrow int \rightarrow int\ list$ is a representation function, yielding a list as the denotation for a vector represented implicitly by an array p.v and its length p.vlen. Representation functions always refer to the state Σ, a location, and a tuple of C scalar values (of type Val^n). As they occur frequently, our language provides *representation anti-quotations* to express them easily: a representation function $R : \Sigma \rightarrow Loc \rightarrow Val^N \rightarrow \alpha$ can be refered to inside a quotation as ^R{x_0, x_1, .., x_N}, where the x_i are C lvalues. Fig. 6 shows a matrix inversion operation specified using representation anti-quotations.

```
/*@
  @requires \valid(m) && \valid(inv) &&
    ${ invertible ^Matrix{m} }
  @modifies *inv
  @ensures \result != -1 -->
    ${ ^Matrix{m} * ^Matrix{inv} = (1 :: mat3) }
  @*/
int invert_transform(const matrix3 *m, matrix3 *inv);
```

Fig. 6. Example specification: matrix inversion. m and inv are not required to be distinct. The specification assumes that a type of 3×3 matrices *mat3* and a constant $Matrix : \Sigma \to Loc \to mat3$ are defined in Isabelle. *1* is an overloaded constant, used here for the type *mat3*.

An alternative to using representation functions in our setting would be to develop a component-based state model that can be used in specifications directly (as in [5]). In our opinion, no state model can be conceived that is generic, yet makes it comfortable to directly work with representations of C values as provided by that model. In particular, this would require that all domain theorems are formulated in terms of the state model, which is unrealistic. Further, not all objects can be referenced as lvalues (in particular, there is no expression evaluating to a whole array).

4 Generating Verification Conditions

In this section we describe how specifications are translated to theorems in Isabelle, how these theorem are proven, and what automatic support we provide.

4.1 Translation to correctness propositions

All translated annotation elements given for a function specification are composed to form a proposition whose validity entails that the function at hand (call it f) satisfies its specification. This proposition basically is a Hoare triple as of Sec. 2.6. To be exact, it states that under the assumption that *all functions called by* f do satisfy their specification, execution of f in an arbitrary pre-state satisfying the precondition will, for all possible arguments of the right type and arity, *(1)* terminate, *(2)* not alter objects except those mentioned in the modification set, *(3)* not access arrays outside their bounds,*(4)* not dereference invalid (or **NULL**) pointers,*(5)* not perform a division by zero, and *(6)* end in a state that together with the pre-state satisfies the postcondition. Note that this proposition is formulated over the semantic interpretation of a C function, in which we abstract the numeric types. It is therefore possible to write code that will verify, but display unwanted behaviour in practice, by combining floating-

point arithmetic and pointer manipulation.[2] We argue, however, that this kind of interplay is not common in applications, and can be avoided by other means.

Formally, the correctness proposition is as follows. Consider the specification

```
/*@ @requires Pre    @modifies mlist    @ensures Post @*/
int f(double x)
```

Let Γ be an appropriate environment which contains the relevant global variables and specifications of f's callees; further let *arg_types p a* express that the list of actual parameters a matches the formal parameter list p w.r.t. their types. Then the specification gets translated to the following Isabelle proposition:

$$\begin{aligned}\forall\ \Lambda\ args\ S_1 \bullet \Lambda = [\![\mathsf{mlist}]\!]\ \Gamma\ S_1 \wedge arg_types\ [\mathsf{x}]\ args \longrightarrow \\ \Lambda, \Gamma \vdash_f\ [\lambda S.\ S = S_1 \wedge \#(\mathsf{Pre})\ \Gamma\ S\ [\mathsf{x}]] \mathsf{f}(\mathsf{double}\ \mathsf{x}) \\ [\lambda(r,S).\ \#(\mathsf{Post})\ \Gamma\ (S_1, r, S)\ [\mathsf{x}]]\end{aligned} \quad (2)$$

4.2 Program Proof Rules

To derive verification conditions for a concrete program and specification, we perform a backwards proof. Starting with proposition (2), we match the corresponding rule on the current state. By reducing the program term, we build up the postcondition. For this, there has to be at least one rule for each constructor of the datatypes representing the language, and the rules have to be formulated in such a way that the postcondition of the conclusion is a single variable, so it can match on any postcondition. The proof is performed by an Isabelle tactic that transforms the initial proof obligation (2) into a single *intermediate verification condition* (iVC) by applying proof rules that subsequently reduce the program term to purely logical expressions. In total, we have $\sim$ 80 rules for function definitions, local declarations, blocks, statements, expressions and all constituent parts of these. Fig. 7 shows three of the rules.

As usual, the rules are best read from bottom to top. Rule (IntLVal) reduces an lvalue integer expression (whose value is an integer) to the lvalue itself (whose value is the location of the integer). This is done by reflecting the action of reading a location within the predicate: the postcondition Q expecting the integer value becomes $(\lambda lv\ S \bullet \mathbf{let}\ iv = read_int\ lv\ S\ \mathbf{in}\ Q\ iv\ S)$, which expects a location, reads it as an integer, binds that value to an intermediate variable iv and then passes iv to Q. The let-binding avoids a blowup in predicate size and keeps the number of read operations in predicates small.

Since our pre- and postconditions are boolean-valued functions, we cannot use substitution to reflect assignments in predicates. Instead, we explicitly modify the program state. Rule (Assign) shows this: to prove an assignment with

[2] The following code snippet will cause a segmentation fault on an IA-32 system, but is verifiable since `i <= 10000 && d > 0` is an invariant in the semantic interpretation.

```
int i = 0; int * p = &i; double d = 1.0;
while (i++ < 10000) d /= 2.0;
if (d <= 0) p = NULL;
*p = 0;
```

$$\frac{\Lambda, \Gamma \vdash_{lv} [P]\ \mathsf{lval}\ [\lambda lv\ S \bullet \mathbf{let}\ iv = (read_int\ lv\ S)\ \mathbf{in}\ Q\ iv\ S]}{\Lambda, \Gamma \vdash_i [P]\ \mathsf{lval}\ [Q]} \quad \text{(IntLVal)}$$

$$\frac{\begin{array}{c}\forall l \bullet (\Lambda, \Gamma \vdash_e [R\ l]\ \mathsf{t}\ [\lambda a\ S \bullet \mathbf{let}\ T = update\ l\ a\ S\ \mathbf{in}\ Q\ T]) \\ \Lambda, \Gamma \vdash_{lv} [P]\ \mathsf{lv}\ [\lambda l\ S \bullet R\ l\ S \wedge l \in \Lambda]\end{array}}{\Lambda, \Gamma \vdash_s [P]\ \mathsf{lv} = \mathsf{t}\ [Q]} \quad \text{(Assign)}$$

$$\frac{\begin{array}{l}\forall \Lambda'\ S\ N \bullet\ \ \Lambda' = [\![\mathsf{mlist}]\!]\,\Gamma\,S \longrightarrow \\ \qquad (\Lambda', \Gamma \vdash_s [J\,\Lambda'\,S\,N]\ \mathsf{c}\ [\lambda T \bullet invar\,\Gamma\,T \wedge [\![\mathsf{var}]\!]\,\Gamma\,T < N] \wedge \\ \qquad \Lambda', \Gamma \vdash_b [K\,\Lambda'\,S\,N]\ \mathsf{b}\ [\lambda b\ T \bullet (b \longrightarrow J\,\Lambda'\,S\,N\,T) \wedge (\neg b \longrightarrow F\,T)])\end{array}}{\begin{array}{l}\Lambda, \Gamma \vdash_s [\lambda S \bullet \mathbf{let}\ M = [\![\mathsf{mlist}]\!]\,\Gamma\,S\ \mathbf{in} \\ \qquad (invar\,\Gamma\,S \wedge M \subseteq \Lambda \wedge \\ \qquad (\forall T \bullet S \simeq_M T \longrightarrow invar\,\Gamma\,T \longrightarrow K\,M\,S\,([\![\mathsf{var}]\!]\,\Gamma\,T)\,T))] \\ \quad \mathsf{while\ b\ c\ (loopanno}\ invar\ \mathsf{mlist\ var)} \\ \quad [F]\end{array}} \quad \text{(WhileTotal)}$$

Fig. 7. Proof rules for integer lvalues, assignments and while statements

postcondition Q, we evaluate the lvalue we assign to (in the second premiss), showing it is a modifiable location ($l \in \Lambda$). We then evaluate the expression t (in the first premiss); the updated state is bound to an auxiliary variable T, which is passed to Q. This is equivalent to substitution: we create the predicate stating that updating the state at lv with t yields a state satisfying Q. The state predicate R is both the precondition of the first premiss and the postcondition of the second, thus logically connecting Q and P in the conclusion.

We shall only illustrate the rule for loops (WhileTotal) here. It demonstrates that proof rules for real program verification are a little more complex than what idealised textbook variants might suggest, and shows why it is useful to be able to prove the rules formally correct, making a manual correctness inspection of rules like these unnecessary. The precondition of the conclusion requires that the annotated invariant *invar* holds; that the annotated set of modifiable locations, M, is a subset of the modifiable location set Λ in the context; and that in each state T with $S \simeq_M T$ we may infer K from the invariant. We want to show that F holds after execution of the while statement. This holds given two premises. The first premiss states that an arbitrary run of the body in a state satisfying J re-establishes the invariant *invar*. The second premiss states that after evaluation of the condition b under the precondition K we either obtain F directly — for the case where b evaluates to *false* – or we end in a state satisfying some intermediate predicate J, if b evaluates to *true*. Both premises are formulated in the context of the annotated modification set mlist, instead of the context Λ of the loop itself, ensuring the loop body only modifies locations as annotated in the loop specification. Termination of the loop is also ensured, employing the annotated variant var; without going into details, the rule encodes the requirement that the variant, interpreted as a natural number, strictly decreases in each iteration.

Weakened weakening through modification sets. Modification sets introduce the essential property of *framing* (see Sec. 2.6), which shows up in what we call *weakened weakening*. In the conclusion of rule (WhileTotal) we weaken the invariant, roughly: $\forall T \bullet S \simeq_M T \longrightarrow invar\, T \longrightarrow K\, T$. This is essentially the statement that the invariant implies the weakest precondition of the loop body w.r.t. the invariant itself, K. We do not need to be able to do this for *any* state, but only for those states T with $S \simeq_M T$. Thus, facts about S can be used in proving the weakening, since these are also valid for the quantified states T if they are independent of the locations M. E. g., if the loop is the first statement of the function to be verified, then we know the function's precondition holds in S. The invariant therefore only needs to specify those properties that concern locations in M, which is exactly the desired framing property.

4.3 Reduction to domain-related and program safety VCs

The iVC is a single logical expression whose validity ensures program correctness. This expression is simplified through a set of tactics which ultimately yield verification conditions of three kinds, which then need to be proven interactively. The first ones are domain-related VCs, e.g. that a function specified to compute the inverse of an affine transform actually does so. The second ones are program safety VCs that could not automatically be proven. These are comprised of non-trivial array bounds checks, where the array is indexed in other ways than by an iteration variable, pointer dereferencing checks and checks for division by zero. Since pointer arithmetic is hardly used in mathematical operations, the safety of pointer dereferencing can be proven automatically by the tactics in most cases. The third kind of VC is concerned with modification sets; these VCs demand that only the specified locations have been modified. The only VCs of this kind that cannot be proved fully automatically are, again, those involving non-trivial array indexing. These are seldom and mostly easy to prove manually, since an assumption about the validity of the relevant array access will be available due to an earlier proof of that fact.

There are three main technical features of these tactics worth mentioning. *(1)* Thanks to the structure of the iVC, where every intermediate value and state is let-bound, we can avoid a combinatorial explosion in the size and number of VCs, similar to [15], because complex expressions do not occur repeatedly. *(2)* Concerning aliasing, we have proven ~ 100 lemmas about our state model that allow to exploit a restricted property of the split heap model [8], namely that structure fields with different names cannot be aliased for properly aligned structures. (a−>f and b−>g always denote different locations for valid pointers a and b). This eliminates many unnecessary VCs that would normally arise in a basic state model like ours. *(3)* Finally, properties of representation functions are known to the simplification tactics, such that, e. g., an update on a **struct** Point in the code is reflected by an associated update on the model point in the VC. Likewise, equalities such as $R\,(update\, l\, v\, \Sigma)\, m = R\, \Sigma\, m$, expressing that the R-representation at location m is independent of updates at l (with appropriate conditions on m and l, of course) are built into the tactics.

```
/*@ @requires \separated(ps, len, rs, rs_len)
   && \separated(qs, len, rs, rs_len)
   && len <= rs_len && -$pi/2 <= alpha && alpha <= $pi/2
  @modifies rs[0:len].x, rs[0:len].y
  @ensures (\result == OK) -->
  ${ ALL (s::real). (ALL i. 0 <= i & i < `len -->
       arc_end s `alpha ^Point{ps[$i]} = ^Point{qs[$i]}) -->
  (ALL i. 0 <= i & i < `len -->
     arc s `alpha ^Point{ps[$i]} <= convex_hull
      {^Point{ps[$i]},^Point{rs[$i]},^Point{qs[$i]}})
   } @*/
status archull(double alpha, vec2d *ps, vec2d *qs, int32 len,
                          vec2d *rs, int32 rs_len );
```

Fig. 8. Example specification: the `archull` function

4.4 Example: Verification at Work

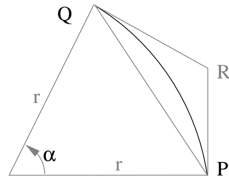

Fig. 9. The arc hull

Fig. 8 shows the specification of a function archull which calculates convex hulls of arcs. Each arc is given by its endpoints P_i and Q_i and the opening angle α of the corresponding circle segment which uniquely determines the radius r. The convex hull is constructed with the intersection point R_i of the tangents through P_i and Q_i, see Fig. 9. This is a typical function of medium complexity, which mixes domain-related and technical requirements. The function takes an array of start- and endpoints, and a single angle α, and stores the R_i in a result array.

The function is about 40 lines of C code, with calls to five other functions. The translated Isabelle theory is about 500 lines. Our tactic reduces this function to 4 domain-related proof obligations, 8 technical and program safety obligations, and 10 pointer-validity obligations (which are proven automatically). [3]

5 Related Work And Conclusion

This paper has presented an approach to the verification of C programs in the context of software certification in the area of mobile robotics. Its distinctive features are a deep embedding of a subset of C into Isabelle, and specification by annotation in a language directly based on Isabelle's higher-order language.

Closely related approaches such as Frama-C [17], Caduceus [3] or JML have a comparatively weak, essentially first-order specification language, which in turn can be used with many prover backends. In contrast, we have an expressive,

[3] The relevant Isabelle theories are provided at `http://www.informatik.uni-bremen.de/~cxl/sources/fm09.tgz`; a public release of the tool will be made avaibable at `http://www.sams-project.org/`.

higher-order language, geared towards a specific prover. We believe the added expressivity of higher-order logic compensates for the loss of versatility. By that we appeal to both conciseness of specifications (hence readability) and logical expressivity. We cannot imagine how one would give a functional specification of, e.g., geometric algorithms in pure first-order logic. Another difference is the direct embedding of the programming language, as opposed to using an intermediate language such as Simpl [5] or Why [3]. Thus, we have to rely less on the transformations performed by a syntactical front-end, increasing confidence in the correctness of the verification process.

There is other work using theorem provers (including Isabelle) to verify C programs in different application domains, such as the L4 Verified project [18] verifying an operating system kernel, Verisoft [19] concerned with comprehensive verification (from the hardware to applications, including a verified compiler), etc. The different application domains emphasize that there must be different tools for different application scenarios [20] as each have their own requirements. Moreover, it makes a big difference whether verification is used for external certification, debugging or quality assurance. For example, model-checkers are far more useful for debugging (e.g. [21]) than for certification.

Refinement calculi like VDM and Z are close to our approach concerning the rich mathematical language used. However, we do not follow a refinement approach, although this is feasible in Isabelle. Instead, the concrete source code that will run on the real system comes under formal scrutiny, which is particularly relevant for safety-critical systems.

A denotational semantics (as opposed to an operational one, which captures the C standard more closely [22]) has not only the advantage of easier verification of the proof rules, but we can also use the denotations in the specification.

Our framework can presently handle C functions of medium length. The limiting factor is the size of the proof state produced during the generation of the verification conditions, which is—as usual for verification condition generators—exponential in the number of sequential conditional branches and linear for other program constructs. Care has been taken to keep the number of generated VCs small. A join construct helps to keep the exponentional growth incurred from conditionals in check. To be able to verify longer functions, one breaks them down into smaller components. Presently, the framework consists of the frontend, which is 14 kloc of Haskell (including a checker for MISRA conformance), and the Isabelle backend. This contains 30 theories with a total of 1150 theorems. The tactical support is 1700 lines of SML code. Using the prover language for specification has the further advantage that it allows a comprehensive formal approach, from domain modelling down to the code in one formalism [16].

The approach has been presented to the certification authority (TÜV Süd), and preliminarily approved. The final presentation of the project results is scheduled for October 2009. The current experience is positive. The design of the specification language has been validated in the weekly code and specification reviews within the project, where researchers with no previous exposure to formal methods and Isabelle were able to grasp specifications such as Fig. 8 quickly.

References

1. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: Formal Methods for Components and Objects. LNCS 4111, Springer (2006) 342–363
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C specification language. `http://frama-c.cea.fr/download/acsl_1.4.pdf` (October 2008) Preliminary design, version 1.4.
3. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Proc. of the 19th Int. Conference on Computer Aided Verification (CAV '07). LNCS 4590, Springer (2007)
4. Nipkow, T.: Hoare logics in Isabelle/HOL. In Schwichtenberg, H., Steinbrüggen, R., eds.: Proof and System-Reliability, Kluwer (2002) 341–367
5. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
6. Winskel, G.: The Formal Semantics of Programming Languages. Foundations of Computing Series. MIT Press (1993)
7. Programming languages — C. ISO/IEC Standard 9899:1999(E) (1999) 2nd Ed.
8. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction. (2000) 102–126
9. MISRA-C: 2004. Guidelines for the use of the C language in critical systems. (2004)
10. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1) (1991) 55– 92
11. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. PLDI '03, San Diego, California, USA, ACM Press (2003) 196–207
12. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. Software Engineering, IEEE Transactions on **21**(10) (1995) 785–798
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS 4963, Springer (2008) 337–340
14. Barrett, C., Tinelli, C.: CVC3. In: Proc. of the 19th Int. Conference on Computer Aided Verification (CAV '07). LNCS 4590, Springer (2007) 298–302 Berlin, Germany.
15. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Proc. POPL '01, New York, NY, USA, ACM Press (2001) 193–205
16. Frese, U., Hausmann, D., Lüth, C., Täubig, H., Walter, D.: The importance of being formal. In Hungar, H., ed.: Int. Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert '08). To appear in Electronic Notes in Theoretical Computer Science (2008)
17. Frama-C. Website at `http://frama-c.cea.fr/` (2008)
18. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: Taking microkernels to the next level. ACM Operating Systems Review **41**(4) (2007) 3–11
19. The VeriSoft project. Web site at `http://www.verisoft.de/`
20. Lamsweerde, A.v.: Formal specification: a roadmap. In: ICSE '00: Proc. of the Conference on The Future of Software Engineering, New York, NY, USA, ACM (2000) 147–159
21. Ball, T., Millstein, T., Rajamani, S.K.: Polymorphic predicate abstraction. ACM TOPLAS **27**(2) (March 2005) 314– 343
22. Norrish, M.: C Formalised in HOL. PhD thesis, University of Cambridge (1998)