2017 IEEE 18th International Symposium on High Assurance Systems Engineering

High Assurance Code Generation for Cyber-Physical Systems

Tze Meng Low, Franz Franchetti
Carnegie Mellon University
{lowt, franzf}@cmu.edu

***Abstract*—High Assurance SPIRAL (HA-SPIRAL) is a tool that synthesizes a faithful and high performance implementation from the mathematical specification of a given controller or monitor. At the heart of HA-SPIRAL is a mathematical identity rewrite engine based on a computer algebra system. The rewrite engine refines the mathematical expression provided by a control engineer, through mathematical identities, into an equivalent mathematical expression that can be implemented in code. In this paper, we discuss the use of HA-SPIRAL in generating provably-correct and high-performance implementations for different controllers and monitors for autonomous land and air vehicles.**

I. Introduction

The challenges of producing software with a high degree of dependability (high assurance software) for control code of cyber-physical systems such as autonomous vehicles have given rise to a number of formal approaches that address different parts of the problem. Model checking and hybrid verification tools provide proofs that a model of a system behaves as specified and that a control system is stable. Proof assistants (Coq[1], Isabelle[2], PVS[3]) and verified compilers (CompCert[4]) can provide a proof of certain properties of the code and establish the equivalence between source code and the compiled binary. However, the synthesis of high-performance, floating-point heavy control code from a high level specification together with the guarantee that the code faithfully implements the behavior of the specification relative to arithmetic over the real numbers is relatively unaddressed.

High assurance SPIRAL (HA-SPIRAL) is a code generation tool developed to address this problem of generating provably correct implementations from the mathematical specifications of the desired control code [5]. Starting with the mathematical specification/formulation of the controller/monitor, HA-SPIRAL produces both a highly optimized implementation of the specification, and a trace that provides evidence that the implementation is mathematically equivalent to the specification. The generated code can be compiled with certified compilers to provide an unbroken chain of evidence that the final binary is a faithful implementation of the original specification.

In this paper, we demonstrate the use of high assurance SPIRAL (HA-SPIRAL) within the context of implementing monitors and controllers for autonomous air and ground vehicles.

II. HA-SPIRAL and Code Generation

Control engineers typically describe their algorithm using mathematical equations or mathematical operators that have specific semantic meaning. These meanings are often lost during the implementation process of turning the mathematical description of the algorithm into code. HA-SPIRAL addresses this difference in representations by providing a domain specific language, the Hybrid Control Operator Language (HCOL), to control engineers that is more akin to the mathematical formulation commonly used to described control algorithms. Through a series of identity rewriting, HA-SPIRAL rewrites the input mathematical expression into a more detailed but mathematical equivalent expression that can be transformed into code.

A. Geo-Fence Monitor Example

We illustrate the use of HA-SPIRAL with the example of a geo-fencing monitor that checks if a vehicle remains within a specified area. The location of the vehicle can be modeled as a point, and the geo-fenced area can be monitored as one or more convex polygons. The vehicle is said to be in the geo-fenced area if the point is inside at least one of the polygon.

For simplicity, we consider the case where the geo-fence can be described by a single convex polygon. Mathematically, this can be described as follows:

$$\text{InsidePoly}(x) = (Ax - b) \leq \vec{0},$$

where matrix A, and vector b are constants describing the polygon, $\vec{0}$ is a vector of all zeros, and x is a vector describing the position of the vehicle. In essence, the geo-fence can be computed in four separate steps: 1) a matrix-vector multiplication of A and x, followed by 2) a vector subtraction with b, 3) a point-wise operation that compares all elements of the resulting vector against zero, and 4) a reduction operation is performed on the results of the comparisons, where the results are reduced via conjunction to obtain a single boolean value that tells us if the position of x is within the polygon described by the matrix A and vector b.

An implementation of the monitor could be as follows:

```
for (int i = 0; p != m; ++i){
  int out = 0;      float tmp =  0.0;
  for (int j = 0;  != n; ++j)
      tmp += A[i][j] * x[j];
  out = out && ((tmp - b[i]) <= 0);
}
return out;
```

1530-2059/17 $31.00 © 2017 IEEE
DOI 10.1109/HASE.2017.28

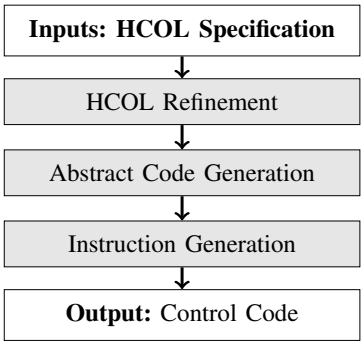

Fig. 1. HA-SPIRAL system consists of multiple rewriting phases. Each phase refines the input specification into a more detailed expression. The final phase uses rewrite rules to specialize the mathematical expression into code for a specific platform.

Notice that mathematical operations such as the matrix-vector product, vector-vector subtraction, comparison and reduction operations are no longer obvious from the implementation. Steps 2 through 4 of the geo-fence algorithm have been merged via Loop Fusion/Merging, a traditional compiler loop optimization [6]. This makes verifying the correctness of the code, i.e. the faithful implementation of the original specification, difficult.

B. HA-SPIRAL's rewriting system

HA-SPIRAL uses a set of rewrite rules based on mathematical identities to transform one representation to another. Given an input specification written in HCOL, HA-SPIRAL repeatedly applies rewrite rules that refines the input specification, into more expressive mathematical expression that can be generated into code. As all rewrite rules in HA-SPIRAL are mathematical identities, this means that the expressions before and after the application of a rewrite rule can be proven to be equivalent. By implementing HA-SPIRAL and HCOL on top of the computer algebra system, GAP [7], the correctness of the rewrite rules are enforced by the mathematical rules built within the system.

We illustrate the different phases of rewriting within HA-SPIRAL (as shown in Figure 1) to turn an input specification into output code with the first step of the geo-fence example, the matrix-vector product.

Phase 0: HCOL Specification. Recall that the key reason for HCOL is to provide control engineers a language that is similar to the mathematics they would write. As such, HCOL is defined in terms of operators, operations that are applied to the input. In addition, these operators are higher level mathematics operations, i.e., they are usually easy to specify mathematically, but would typically take multiple lines of code to implement. An example of such an operator is the matrix-vector product, $y = Ax$, used in the first step of our geo-fence example.

The input to HA-SPIRAL is the operator:

$$\mathrm{MatrixVectorProd}_{m,n,(a_i)}\,.$$

Subscripts m, n and (a_i) are parameters to the operator. In this case, m and n describe the dimensions of the matrix A, and (a_i) represents the specific row values of A.

Phase 1: HCOL Refinement. The matrix-vector product,

$$y = Ax,$$

where A is a $m \times n$ matrix and x, y are vectors of length n and m respectively. y can be computed by first partitioning y, and A into rows, i.e.,

$$y \to \begin{pmatrix} \psi_0 \\ \hline \vdots \\ \hline \psi_{m-1} \end{pmatrix} \text{ and } A \to \begin{pmatrix} a_0^T \\ \hline \vdots \\ \hline a_{m-1}^T \end{pmatrix}$$

and then each element of y (ψ_i) is computed by performing a scalar product of the corresponding row of A with x, i.e.

$$\begin{pmatrix} \psi_0 \\ \hline \vdots \\ \hline \psi_{n-1} \end{pmatrix} = \begin{pmatrix} a_0^T x. \\ \hline \vdots \\ \hline a_{n-1}^T x \end{pmatrix}$$

This mathematical identity is captured within HA-SPIRAL as a rewrite rule of the form:

$$\mathrm{MatrixVectorProd}_{m,n,(a_i)} \to \left[\begin{smallmatrix}\vdots\end{smallmatrix}\right]_{i=0}^{m-1} \mathrm{ScalarProd}_{n,a_i},$$

where $\mathrm{ScalarProd}$ represents the operator for the scalar product operation, and the operation

$$\left[\begin{smallmatrix}\vdots\end{smallmatrix}\right]_{i=0}^{m-1}$$

can be thought of as the vertical stacking of multiple $\mathrm{ScalarProd}$. A more formal definition is defined subsequently.

The scalar product operator can be further decomposed based on the identity

$$x^T y = \sum_{i=0}^{n-1} \chi_i \psi_i,$$

into a point-wise multiplication all n elements of x and y, followed by a reduction (summation) of the results from the multiplication. In HCOL, this identity is captured by the rewrite rule

$$\mathrm{ScalarProd}_{(\alpha_i)} \to \mathrm{Reduction}_{n,(\alpha,\beta)\mapsto(\alpha+\beta),0} \circ \mathrm{Pointwise}_{n,(x_i,\alpha_i)\mapsto\alpha_i x_i},$$

where $\circ$ represents operator composition, similar to functional composition $(f \circ g)(x) = f(g(x))$.

Other rewrite rules within HA-SPIRAL that are used in our geo-fence model are captured in Figure 2.

Phase 2: Abstract Code Generation. At the end of the rewriting, HA-SPIRAL produces a mathematical expression

involving simple arithmetic/comparison functions that can either be directly translated into code, or replaced with an abstract code template. In this particular phase, mathematical expressions that operate on higher dimensional data (e.g. vectors and matrices) are transformed into loops over scalar computations. Exposing scalar computations may also introduce opportunities for optimizations, which are exploited in this phase on the code generation process.

Recall that in order to compute our matrix-vector mutliplication, the output of every scalar products needs to be placed in a different memory address as described by the operator

$$\left[\begin{matrix}\vdots\end{matrix}\right]_{i=0}^{m-1} \text{ScalarProd}_{n,a_i}.$$

We leverage the fact that a consecutive block of memory can be thought of as a vector, where each element is a unique memory address. This means that selecting elements from the vector can be done via the multiplication of the memory vector with the appropriate standard basis vectors. For example, the following mathematical expression gathers/selects the first n elements (i.e., the first row) from the matrix A,

$$(e_1^{mn} + e_2^{mn} + \ldots + e_n^{mn})^T A,$$

where e_i^{mn} is a vector of length mn, and has a one in the i^{th} position, and zeros in all other positions, in order to compute the first scalar product.

Storing elements into the memory vector can be performed using a similar mathematical expression. For example, the output of the j^{th} scalar product, denoted as α_j, can be stored back into the appropriate location in the memory vector, y with the following expression

$$(e_j^m)^T y = \alpha_j.$$

These functions that selects and stores elements to and from memory are automatically generated via HA-SPIRAL based on the input and output dimensions of the operator, which in turn are governed by the rules of mathematics.

Phase 3: Instruction Generation. In this phase, the abstract code generated in the previous phase is specialized to a particular output language for a particular architecture.

C. Hybrid Control Operator Language (HCOL)

Rewrite rules within HA-SPIRAL are written in the Hybrid Control Operator Language (HCOL) that was designed to be extensible so that higher level mathematical constructs can be built from operators that were previously formalized in HCOL.

HCOL is an operator-based language where the primitives in the languages are operators and operations. Operators within HCOL are essentially mathematical functions, while operations are higher order operators that take one or more operators and return a new operator. The core of HCOL is built upon a small number of basic operators and operations highlighted in the rest of this subsection.

Operator. In HCOL, the following primitive operators are defined.

1) *Constant Operator.* The Constant Operator, denoted C_c, returns the constant value c, regardless of its input, i.e.
$$C_c(x) = c$$
2) *Identity Operator.* The Identity operator, denoted $I(x)$, simply returns the input, i.e.
$$I(x) = x$$
3) *Atomic Operator.* The Atomic Operator is any mathematical function of n-arity, and is denoted as
$$f_n(x_0, x_1, \ldots, x_n) = y$$
4) *Pointwise Operator.* The Pointwise Operator is a generalization of the Atomic Operator where n Atomic operators f_i's are applied to each of the n elements, x_i of a vector input to compute a vector output. This is similar to the `map` function in many functional languages.
We denote the Pointwise operator using the following symbol:
$$\text{Pointwise}_{n,f_i}$$
Note that while we do not restrict that f_i's to be the same operator, it is usually the case that the same operator is applied to every input element.
5) *Reduction Operator.* The Reduction Operator is the HCOL equivalent of the `fold` function in many functional language. Mathematically, it can be described as
$$\begin{aligned}&\text{Reduction}_{n,f_i,c} = \\ &\quad f_{n-1}(x_{n-1}, \\ &\qquad f_{n-2}(x_{n-2}, \ldots f_0(x_0, C_c(.))))\end{aligned}$$
6) *Selection Operator.* Often known as the gather operator, this operator picks out particular scalar elements from a higher dimensional input through the use of basis vectors, i.e.
$$(e_i^n)^T((\chi_0, \ldots, \chi_{n-1})) = \chi_i.$$
7) *Embed Operator.* The embed operator puts a scalar element into a higher dimensional output. Again, the basis vector notation from linear algebra is used to denote the Embed operator.
$$(e_i^n)^T(x) = (*, \ldots, \overbrace{x}^{i^{th}\text{element}}, \ldots, *),$$
where $*$ represents values that are undefined.

Operation. In order to create mathematical operators from the primitive operators, operations, or higher order operators, are defined within HCOL:

1) *Compose Operation.* Given two operators, A and B, the Compose operation, denoted by $\circ$, takes the output of the first operator A and uses it as the input to operator B. In essence, the Compose operation is function composition, and can be described as follows:
$$(B \circ A)(x) = B(A(x))$$

$$
\begin{aligned}
\mathrm{InsidePoly}_{m,n,A,b} &\rightarrow \mathrm{ForAll}_{i=0}^{n-1}(\mathrm{Pointwise}_{n,x_i\mapsto x_i\leq 0} \circ \mathrm{LinearSys}_{m,n,A,b}) \\
\mathrm{LinearSys}_{m,n,A,b} &\rightarrow \mathrm{Pointwise}_{n,(x_i,\beta_i)\mapsto x_i-\beta_i} \circ \mathrm{MatrixVectorProd}_{m,n,A} \\
\mathrm{MatrixVectorProd}_{m,n,(a_i)} &\rightarrow \begin{bmatrix}\vdots\end{bmatrix}_{i=0}^{n-1} \mathrm{ScalarProd}_{a_i} \\
\mathrm{ScalarProd}_{(\alpha_i)} &\rightarrow \mathrm{Reduction}_{n,(\alpha,\beta)\mapsto(\alpha+\beta),0} \circ \mathrm{Pointwise}_{n,(x_i,\alpha_i)\mapsto\alpha_i x_i} \\
\mathrm{ForAll}_{i=0}^{n-1}(A_i) &\rightarrow \mathrm{Reduction}_{n,(\alpha,\beta)\mapsto\alpha\wedge\beta,\top}
\end{aligned}
$$

Fig. 2. Subset of mathematical operators in HA-SPIRAL. More complex operators can be composed from the existing operators within HA-SPIRAL. Reduction and PointWise are special operators in HA-SPIRAL's Hybrid Control Operator Language (HCOL) that are replaced by code templates in order to transform mathematical expressions into code involving loops.

2) *Direct Sum Operation.* The Direct Sum operation, denoted by $\oplus$, takes two operators, A and B and returns an operator that performs operator A on the first part of the input and B on the second part of the input. Mathematically, the Direct Sum is defined as follows:

$$(A \oplus B)\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} A(x) \\ B(y) \end{pmatrix}$$

3) *Broadcast Operation.* Given m operators A_i, the Broadcast operator duplicates the input m times and performs the Direct Sum Operation on the duplicated output, i.e.

$$\left(\begin{bmatrix}\vdots\end{bmatrix}_{i=0}^{m-1} A_i\right)(x) = \begin{pmatrix} A_0(x) \\ \vdots \\ A_{m-1}(x) \end{pmatrix}$$

D. Optimizations

Traditional loop optimizations within HA-SPIRAL are also implemented with the same rule-based system described previously. To illustrate how optimization is performed within HA-SPIRAL, we return to the geo-fence example.

Recall that Step 2 of the algorithm is a vector subtraction of the output of the previous step with the vector b. Mathematically, the output of the vector subtraction (i.e., $y = x - b$) is

$$\begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{n-1} \end{pmatrix} = \begin{pmatrix} \chi_0 - \beta_0 \\ \vdots \\ \chi_{n-1} - \beta_{n-1} \end{pmatrix},$$

where every element is subtracted with the corresponding element from the vector b. In HCOL, the operator is

$$\mathrm{Pointwise}_{n,(x_i,\beta_i)\mapsto x_i-\beta_i}\,.$$

Step 3 of the algorithm checks if every element of the result from Step 2 is less than zero. Mathematically, the following mathematical operation,

$$\psi_i = \chi_i \leq 0,$$

is performed on every input element to Step 3. The net result of applying both steps of the algorithm is an output whose elements are

$$\begin{pmatrix} (x_0 - \beta_0) \leq 0 \\ \vdots \\ (x_{n-1} - \beta_{n-1}) \leq 0 \end{pmatrix}.$$

From the above expression, it should be obvious that operator that is the equivalent to performing Steps 2 and 3, is the pointwise application of the following mathematical expression

$$(x_i, \beta_i) \mapsto x_i - \beta_i \leq 0,$$

where a subtraction with the corresponding element of b is first performed before the result is compared with zero. More concisely, the HCOL operator that is the equivalent of Steps 2 and 3 is

$$\mathrm{Pointwise}_{n,(x_i,\beta_i)\mapsto x_i-\beta_i\leq 0},\,.$$

Notice that this new operator is essentially the function composition of the mathematical expressions for Steps 2 and 3. This new operator represents an optimization because, the original two steps required three passes (once for β_i, and twice for x_i) through the data, whereas this new operator only requires two passes through the data. The above optimization is also known in compiler literature as loop fusion / loop merging.

This optimization is encoded as a more general optimization rewrite rule within HA-SPIRAL:

$$\begin{aligned}\mathrm{Pointwise}_{n,(x_i)\mapsto f(x_i)} \circ \mathrm{Pointwise}_{n,(x_i)\mapsto g(x_i)} \\ \rightarrow \mathrm{Pointwise}_{n,(x_i)\mapsto(f\circ g)(x_i)}\,.\end{aligned}$$

Other optimization rewrite rules within HA-SPIRAL are similarly encoded.

III. Real vs. Floating-Point Numbers

As the input specification is supposed to capture the mathematics used by control engineers, all numbers are implicitly real numbers. However, practical implementations of control codes require that all values be discrete. This necessitates the conversion of all values into one of the IEEE floating point representations. To mitigate this difference in representation, HA-SPIRAL allows one to use interval arithmetics instead of regular arithmetics [8].

$$
\begin{aligned}
x &\rightarrow (x-\epsilon, x+\epsilon) \\
(\ell_0, u_0) + (\ell_1, u_1) &\rightarrow (\ell_0 + \ell_1, u_0 + u_1) \\
(\ell_0, u_0) - (\ell_1, u_1) &\rightarrow (\ell_0 - u_1, u_0 - \ell_1) \\
(\ell_0, u_0) \times (\ell_1, u_1) &\rightarrow (\min(\ell_0\ell_1, \ell_0 u_0, \ell_1, u_0 u_1), \\
&\quad \max(\ell_0\ell_1, \ell_0 u_0 \ell_1, u_0 u_1) \\
(\ell_0, u_0) < (\ell_1, u_1) &\rightarrow \begin{cases} \text{True}, & u_0 < \ell_1 \\ \text{False}, & \ell_0 > u_1 \\ \text{Unknown}, & \text{Otherwise} \end{cases}
\end{aligned}
$$

Fig. 3. Sample rules for implementing interval arithmetic.

For every real number x, HA-SPIRAL replaces x with a tuple of two floating-point values, $(x-\epsilon, x+\epsilon)$, where ϵ is machine precision of the chosen IEEE floating-point representation. This tuple describes the interval where the true value of x may reside. Using the mathematical rules for interval arithmetics, the upper and lower bounds are updated accordingly, thus always ensuring that the true value is always within the interval. Some of these interval arithmetics rules are captured in Figure 3.

A. Comparison and boolean operations with intervals

One of the key differences with the use of comparisons is that there is no notion of equality (or inequality) within interval arithmetics. Recall that each real number is represented by an interval, and the real number can lie anywhere within the interval. This implies that even if two intervals are identical, the represented real numbers may be at different locations within the interval. Similarly, even if the intervals may be different, the real numbers may be the same.

In addition, comparisons over intervals are no longer boolean operations. Instead, they are ternary operators, i.e. there are three possible return values. Consider the following cases when comparing two numbers represented by the tuples (l_0, u_0) and (l_1, u_1), and we want to know if the number represented by (l_0, u_0) is less than the other number:

1) **Always True.** When u_0 is less than l_1, then we know that regardless of the actual real value, the real number represented by (l_0, u_0) will always be less than the number represented by the tuple (l_1, u_1).
2) **Always False.** When u_1 is less than l_0, then we know that regardless of the actual real value, the real number represented by (l_0, u_0) will always be larger than the number represented by the tuple (l_1, u_1).
3) **Unknown.** When $l_0 < l_1$ but $u_0 > u_1$, there exist a region in the interval where the number represented by the first tuple could be larger than the number represented by the second tuple. In addition, there is also a region where the second number is larger. Hence, we cannot be sure if the which of the real numbers are larger.

Finally, boolean operations such as conjunction, disjunction are no longer binary (True or False) but ternary. In particular, the logic system for intervals within HA-SPIRAL is based on Kleene's three-value logic, and is described by the truth-table in Figure 4. In HA-SPIRAL's implementation of Kleene's three value logic, we use the integer values 0, 1, and -1 to represent the values T, F and U, respectively.

$\wedge$	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

$\vee$	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

Fig. 4. Truth-tables for conjunction (left) and disjunction (right) implemented within HA-SPIRAL for boolean operations and comparisons over the interval. T, F, and U represent True, False and Unknown values respectively.

B. Implementation.

Using interval arithmetics will result in at least twice as many floating point operations. This is because the implementation has to compute with both the upper and lower bounds of the interval. However, HA-SPIRAL provides the control engineer the option to implement interval arithmetics using vector or Single Instruction Multiple Data (SIMD) instructions if the platform supports it. As vector instructions allows one to perform multiple identical operations in with a single instruction, implementing interval arithmetics with vector instructions has the effect of speeding up the computation of the interval updates.

The resulting code when implemented using interval arithmetics within HA-SPIRAL is shown in Figure 5. We manually annotated Figure 5 with comments to highlight the sections of generated code that correspond to the code from Page 1. In addition, variable names were manually changed to match the variable names from the code on Page 1. The key take-away from this code example is that implementating interval arithmetic with vector instructions will result in the obfuscation of the non-interval arithmetic algorithm. This makes manual implementation of interval arithmetic a daunting task that is also error-prone. HA-SPIRAL's approach where regular mathematical operators and variables are systematically replaced with their interval arithemtic counterparts can manage this complex process, and is necessary as the program becomes complicated.

IV. Other Control Examples In HCOL

HA-SPIRAL has been used to implement a number of controllers and monitors that have been deployed on actual ground and air vehicle and/or their simulators as part of DARPA HACMS project [9]. In addition, it has been used as both a stand-alone tool, and a back-end code generation engine for the KeYMaeraX hybrid program theorem prover [10]. In this section, we briefly showcase other controllers and monitors that have been implmented using HA-SPIRAL.

A. Dynamic Window Monitor

The dynamic window monitor is a safety critical monitor that checks that the vehicle is still far enough from the nearest obstacle such that the vehicle can still stop without hitting the obstacle [11], [12]. The input to HA-SPIRAL is a monitor

```
#include <immintrin.h>
#include <float.h>

int insideGeoFence(float  *X, float A[5][3], float  *b) {
  static int U1[5];
  __m128d u1;
  int out;
  {
    unsigned _xm = _mm_getcsr();
    _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);

    /* 4 sided polygon */
    for(int i1 = 0; i1 <= 4; i1++) {
      __m128d tmp, u3, x1, x11, x2, x3, x4, x6, x9;

      /* Initialize accumlation variable as an interval */
      tmp = _mm_set1_pd(0.0);

      /*  Position is described in two dimensions*/
      for(int i7 = 0; i7 <= 2; i7++) {

        /* Convert x[j] into an interval */
        u1 = _mm_addsub_pd(_mm_cvtps_pd(_mm_set1_ps(FLT_MIN)), _mm_cvtps_pd(_mm_set1_ps(X[i7])));

        /* Convert A[i][j] into an interval */
        x6 = _mm_addsub_pd(_mm_cvtps_pd(_mm_set1_ps(FLT_MIN)), _mm_cvtps_pd(_mm_set1_ps(A[i1][i7])));

        /* u2 = A[i][j] * x[j] */
        x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x2 = _mm_mul_pd(x1, x6);
        x3 = _mm_mul_pd(_mm_shuffle_pd(x1, x1, _MM_SHUFFLE2(0, 1)), x6);
        x4 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x3, x2));
        u2 = _mm_add_pd(_mm_max_pd(x3, _mm_max_pd(x2, _mm_shuffle_pd(x4, x4, _MM_SHUFFLE2(0, 1)))), _mm_set1_pd(DBL_MIN));

        /* tmp += A[i][j] * [j] */
        tmp = _mm_add_pd(tmp, u2);
      }

      /* x9 = tmp - b[i] */
      x9 = _mm_addsub_pd(_mm_set1_pd(0.0), _mm_sub_pd(tmp, _mm_addsub_pd(
       _mm_cvtps_pd(_mm_set1_ps(FLT_MIN)), _mm_cvtps_pd(_mm_set1_ps(b[i1])))));

      /* U1[i] = x9 <= 0 */
      x11 = _mm_cmple_pd(x9, _mm_set1_pd(0.0));
      U1[i1] = (_mm_testc_si128(_mm_castpd_si128(x11), _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff)) -
    (_mm_testnzc_si128(_mm_castpd_si128(x11), _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff))));

    }
    out = 1;
    for(int i6 = 0; i6 <= 4, ((out != 0)); i6++) {
      float s4;
      s4 = U1[i6];

      /* out = out && U1[i]*/
      out = ((((out == s4))) ? (out) : (((out)*(s4))));
    }

    // BASIC BLOCK BARRIER
    if (_mm_getcsr() & 0x0d) {
      _mm_setcsr(_xm);
      return -1;
    }
    _mm_setcsr(_xm);
  }
  return out;
}
```

Fig. 5. Geo-fence algorithm implemented using interval arithmetics with Streaming SIMD Extensions (SSE) instructions. Comments in the code where manually added, and variable names (when necessary) were manually edited to better highlight the relationship between the code on page 1 and the generated code.

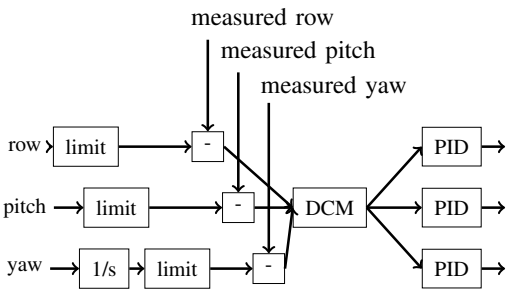

Fig. 6. Block diagram for the attitude controller for the aerial vehicle.

description generated from KeYMaeraX, and can briefly be summarised as

$$\begin{aligned}&\mathrm{SafeDist}_{\alpha,\beta,\gamma,p_o,p_r}(v_r) = \\ &\quad \mathrm{Polynomial}_{\alpha,\beta,\gamma}(v_r) < \|p_o - p_r\|_\infty,\end{aligned}$$

where p_o, p_r represent the location of the vehicle and obstacles, α, β, γ are predefined coefficients related to the dynamics of the vehicle, v_r is the current velocity of the vehicle, and $\mathrm{Polynomial}_{\alpha,\beta,\gamma}$ and $\|.\|_\infty$ are mathematical operators that describe a polynomial of degree two and coefficients α, β, and γ, and the infinity norm of a vector respectively.

B. Attitude Controller for Aerial Vehicle

The attitude controller for an aerial vehicle, in our case a quadcopter, is a controller that stabilizes the vehicle while it is in flight. The controller is essentially made up of PID controllers that have been connected together in order to control the roll, pitch and yaw of the vehicle. The block-diagram of the controller is given in Figure 6. In HCOL, the block-diagram is implemented as separate stages that correspond to 1) computing of the errors in row, pitch and yaw, 2) performing a coordinate transform to the appropriate coordinate frame via a matrix-vector multiplication, and 3) applying PIDs on the transformed coordinates.

$$\begin{bmatrix}\vdots\end{bmatrix}_{i=0}^{3} PID_{gains_i} \circ$$

$$\mathrm{MatrixVectorProd}_{3,3,A} \circ$$

$$\begin{pmatrix} \mathrm{Atomic}_{(x,roll)\mapsto \max(\min(x,r_{max}),-r_{max})-roll} \oplus \\ \mathrm{Atomic}_{(x,pitch)\mapsto \max(\min(x,p_{max}),-p_{max})-pitch} \oplus \\ \mathrm{Atomic}_{(x,yaw)\mapsto \max(\min(x+prev\_yaw,y_{max}),-y_{max})-yaw} \end{pmatrix}$$

C. Statistical Tests

When working with sensors, statistical tests are often required to determine if a sensor is working as expected. An example of such a test would be to perform a z-test to determine if the mean error of a sensor is equal to zero [13]. Mathematically, this z-test is computed as follows

$$P = (\bar{X} - \mu)/\frac{\sigma}{n},$$

where $\mu = 0$, $\bar{X}$ and σ is the sample mean and standard deviation respectively. The HCOL specification that correspond to this z-test for a sample size of n at 95% confidence interval is simply:

$$\begin{aligned}&\mathrm{Ztest}_{n,1.96} = (\mathrm{Mean}_n \,/ \\ &\qquad \mathrm{Atomic}_{x\mapsto\sqrt{x}} \circ \mathrm{Variance}_n \,/n) < 1.96,\end{aligned}$$

where Mean_n and Variance are defined as follows:

$$\begin{aligned}\mathrm{Mean}_n \quad &\rightarrow \quad \mathrm{Atomic}_{x\mapsto x/n} \circ \\ &\qquad \mathrm{Reduction}_{n,(a,b)\mapsto(a+b),0}, \text{ and} \\ \mathrm{Variance}_n \quad &\rightarrow \quad \big(\mathrm{Reduction}_{n,(a,b)\mapsto(a+b),0} \circ \\ &\qquad \mathrm{Pointwise}_{n,x\mapsto x^2}\big) - \\ &\qquad\quad \mathrm{Atomic}_{x\mapsto x^2} \circ \mathrm{Mean}_n,\end{aligned}$$

respectively. The constant 1.96 is the approximate value of the 97.5 percentile (i.e. the α value corresponding to a two-tailed 95% confidence interval).

V. Conclusion

In this paper, we demonstrated the use of HA-SPIRAL in generating a geo-fence monitor that is deployed on an autonomous land vehicle. HA-SPIRAL provides a high level specification language (HCOL) that is similar to the mathematics used by a control engineer to specify the control algorithm. This high level specification is then rewritten through mathematical identities using a rewrite system in order to systematically introduce details and generate code from the high level specification.

While we have only discussed the generation of a provably correct geo-fence monitor, the approach has been used to generate code for a wide range of problems that may be of interest to a control engineer. The synthesized code is highly optimized and takes advantage of modern instruction set extensions, and provides numerical guarantees through the use of interval arithmetic. Together with a verified compiler like CompCert our approach provides an end-to-end guarantee from specification to binary and lends itself to be used as a code synthesis backend for formal verification tools.

Acknowledgment

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0291. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government. No official endorsement should be inferred.

References

[1] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2004, version 8.0. [Online]. Available: http://coq.inria.fr

[2] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[3] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, Jun. 1992, pp. 748–752.

[4] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf

[5] F. Franchetti, A. Sandryhaila, and J. R. Johnson, "High assurance SPIRAL," in *SPIE Defense+ Security*. International Society for Optics and Photonics, 2014, pp. 90 911O–90 911O.

[6] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *In Languages And Compilers For Parallel Computing*. Springer-Verlag, 1994, pp. 301–320.

[7] M. Schönert *et al.*, *GAP – Groups, Algorithms, and Programming – version 3 release 4 patchlevel 4*, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997.

[8] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 661–667. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_52

[9] F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. F. Moura, M. Frnusich, A. Platzer, J. Johnson, and M. Velosa, "High-assurance SPIRAL: End-to-end guarantees for robot and car control," *IEEE Control Systems Magazine*.

[10] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer, "KeYmaera X: An axiomatic tactical theorem prover for hybrid systems," in *CADE*, ser. LNCS, A. P. Felty and A. Middeldorp, Eds., vol. 9195. Springer, 2015, pp. 527–538.

[11] S. Mitsch, K. Ghorbal, and A. Platzer, "On provably safe obstacle avoidance for autonomous robotic ground vehicles," in *Robotics: Science and Systems*, P. Newman, D. Fox, and D. Hsu, Eds., 2013.

[12] S. Mitsch and A. Platzer, "Modelplex: Verified runtime validation of," in *RV*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. A. Smolka, Eds., vol. 8734. Springer, 2014, pp. 199–214.

[13] J. P. Mendoza, M. Veloso, and R. Simmons, "Mobile robot fault detection based on redundant information statistics," October 2012.