High Assurance SPIRAL

Franz Franchetti[a], Aliaksei Sandryhaila[a] and Jeremy R. Johnson[b]

[a]Carnegie Mellon University, Pittsburgh, PA, USA;
[b]Drexel University, Philadelphia, PA, USA

ABSTRACT

In this paper we introduce High Assurance SPIRAL to solve the last mile problem for the synthesis of high assurance implementations of controllers for vehicular systems that are executed in today's and future embedded and high performance embedded system processors. High Assurance SPIRAL is a scalable methodology to translate a high level specification of a high assurance controller into a highly resource-efficient, platform-adapted, verified control software implementation for a given platform in a language like C or C++. High Assurance SPIRAL proves that the implementation is equivalent to the specification written in the control engineer's domain language. Our approach scales to problems involving floating-point calculations and provides highly optimized synthesized code. It is possible to estimate the available headroom to enable assurance/performance trade-offs under real-time constraints, and enables the synthesis of multiple implementation variants to make attacks harder. At the core of High Assurance SPIRAL is the Hybrid Control Operator Language (HCOL) that leverages advanced mathematical constructs expressing the controller specification to provide high quality translation capabilities. Combined with a verified/certified compiler, High Assurance SPIRAL provides a comprehensive complete solution to the efficient synthesis of verifiable high assurance controllers. We demonstrate High Assurance SPIRALs capability by co-synthesizing proofs and implementations for attack detection and sensor spoofing algorithms and deploy the code as ROS nodes on the Landshark unmanned ground vehicle and on a Synthetic Car in a real-time simulator.

Keywords: High-assurance control, sensor fusion, robotics, software verification, program generation

1. INTRODUCTION

Embedded systems represent a ubiquitous foundation of numerous modern technologies. They are used in large-scale data collection and monitoring systems for physical infrastructure control, communication devices, medical devices, ground and aerial vehicles, satellites, and other technologies. Due to design, implementation and functionality conditions and constraints of embedded devices, such as limited computation power and reliance on networking with other devices, they are vulnerable to multiple kinds of attacks, such as direct physical manipulation of devices and corruption of software and device communication.

The research in building high assurance controllers and algorithms that can be verified to meet the required control performance goals is a very active area of work. This activity is sparked by the potential of crippling attacks to critical cyber-physical systems, as epitomized by the Stuxnet worm in 2010. A critical aspect of this research area is the "last mile" problem of the synthesis of high assurance controller implementations to be executed on modern and future embedded system processors in vehicular control systems.

In this paper, we describe a scalable approach to translate high assurance cyber-physical system specifications into highly efficient, platform-adapted, verified control software. Towards this goal, we develop an automatic software generation system called *High Assurance SPIRAL* and the domain-specific *hybrid control operator language* (HCOL). High Assurance SPIRAL takes a controller specification in a control engineer's domain language and translates this specification to a resource-efficient high assurance implementation tuned to a given platform. The system also synthesizes a proof that the implementation is equivalent to the specification. High Assurance SPIRAL is scalable to large problems involving floating-point calculations, provides highly optimized synthesized code, and enables the synthesis of multiple implementation variants to make attacks harder. The system is portable across platforms varying in computational power and architecture. Combined with a verified/certified compiler, it provides a solution to the "last mile" of controller synthesis.

Further author information: (Send correspondence to Franz Franchetti.)
Franz Franchetti: E-mail: franzf@ece.cmu.edu, Telephone: +1 412 268 8297

2. HIGH ASSURANCE SPIRAL

High Assurance SPIRAL is an enabling technology to integrate the wealth of sensing data available on contemporary and future vehicles for achieving high assurance control. It generates highly efficient implementations for an important class of controllers that use physics-based ground truths to detect faults and attacks. The system uses a domain-specific synthesis approach to high assurance controllers described by a high level specification that scales to realistic problem sizes and synthesizes highly efficient implementations together with a proof of equivalency between specification and implementation. Automating the production and verification of such controller implementations enables control engineers to explore diverse approaches and vary the overhead devoted to fault/attack detection to meet resource and real-time constraints. High Assurance SPIRAL synthesizes multiple implementation variants and supports the re-synthesis of the controller when trade-offs change, for instance, due to platform upgrades. The automatic re-synthesis of controllers extends the usability of embedded systems over their life cycle as they get upgraded to more powerful sensing and computing platforms to enhance their capabilities.

The performance of the code synthesized by High Assurance SPIRAL is comparable to the best painstakingly hand-implemented code with the same functionality (in those cases when human-written expert code exists). Furthermore, High Assurance SPIRAL starts from a clean slate. It builds on the SPIRAL autotuning and program generation system[1–5] and extends it to hybrid control systems and high assurance methodologies. SPIRAL is the only system translating a problem specification (given in a mathematical domain-specific language, DSL) into highly efficient code across a wide range of platforms, on par with the best available code. Target platforms range from cell to supercomputers, including GPUs and FPGAs. SPIRAL uses a domain-specific formal translation framework based on mathematics[6–9] that allows the code generation process to prove symbolical equivalence between the specification and internal representations. It contains formal knowledge bases for algorithms (hundreds of journal papers worth of signal processing algorithms), computer architectures, and program generations. SPIRAL's constructive approach to search through term rewriting, backtracking, and constraint satisfaction enables autotuning, algorithm space exploration, and architecture/algorithm co-design. For signal processing applications, SPIRAL has led to industry-grade code generation: SPIRAL-generated functionality is included in Intel's performance libraries (IPP and MKL).

Related work. Systems and technologies similar to High Assurance SPIRAL have been proposed before in different research domains. *SAT/SMT based software synthesis* systems, such as like Sketch[10] and Paraglide[11] automatically synthesize software according to a specification. However, despite tremendous progress in SAT/SMT solving, these systems are limited to moderate-sized (if complicated) pieces of code and mainly support integer code. Research in *Domain-specific languages (DSLs)* has usually focused on how to design the languages, how to compile them, and how to build host systems that enable fast development of new DSLs. Performance, low-level code generation, and symbolic verification/proofs of the generated code are usually not addressed. *Autotuning and program generation* systems, including FFTW,[12] ATLAS,[13] OSKI,[14] FLAME,[15] and TCE, explore search spaces and are capable of producing many variants for the same algorithm, searching for good implementations, often employing domain-specific compilation techniques, but are usually restricted to linear algebra or signal processing kernels. Algorithmic correctness and empirical code testing is usually part of these systems, but formal verification or proof generation for the produced code is not addressed. *Model Checking, automated theorem proving and proof verification* prove that a model of a system meets its specification and that a formula is correct in a given formal system. However, the state-of-the-art is not directly applicable to control code that contains floating-point arithmetic, random processes and data, and does not scale efficiently to large problem sizes. Further, these systems do not address how to translate the specification into its implementation, that is, how to automatically synthesize the implementations.

3. CODE SYNTHESIS AND PROOF GENERATION

In this section, we provide a simple example of a dynamic model-based fault/attack detection possible on an autonomous vehicle to illustrate how classical sensing, control, and dynamics can be represented faithfully and naturally in our HCOL. We caution the reader that this is a simplified illustrative example only.

Figure 1 shows the high-level flow of code/proof co-synthesis. Higher-level approaches to verify algorithms like model checking or differential logic provide verified control algorithms and the respective proofs at that level. Signal-processing or statistical approaches provide tests to establish sensor self-consistency or detect inconsistency. The focus of this paper,

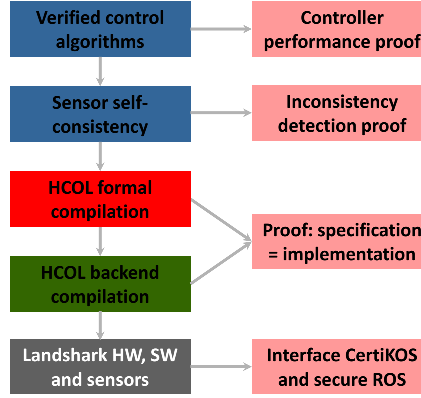

Figure 1: Code/proof co-synthesis in *High Assurance SPIRAL*.

formal compilation translates specifications of such checks into provably equivalent floating-point implementations. A backend compilation stage performs final optimizations before the code is deployed on a robot like the Black-I Robotics Landshark* using a secure operating system like CertiKOS.

3.1 Illustrative Example

First order dynamic model. We assume that the vehicle has two 3-vector sensors: position $\mathbf{x} = (x, y, z)^\top$ and velocity $\mathbf{v}(v_x, v_y, v_z)^\top$ in a global coordinate system. These sensor readings are derived from the actual physical sensors through sensor fusion. In our example we use the redundancy between continuous reading of speed and location sensors to check a simple first order model (velocity $\mathbf{v}(t)$ is the derivative of location $\mathbf{x}(t)$, $\mathbf{v} = \nabla\mathbf{x}$).

We assume that the sensors are read at a fixed sampling interval h. Using the Euler formula we can approximate the position at time $t+h$ as function of the position $\mathbf{x}^t$ at time t and the velocity measurements $\mathbf{v}^{t+h}$ at time $t+h$. We obtain $x^{t+h} \approx x^t + hv_x^{t+1}$; y^{t+h} and z^{t+h} are computed analogously. Similarly, we obtain the velocity at time $t+h$ as function of the position $\mathbf{x}^t$ at time t and $\mathbf{x}^{t+h}$ at time $t+h$ through numerical differentiation: $v_x^{t+h} \approx (x^{t+h} - x^t)/h$. This is the first order approximation that we use for the purposes of an illustration here. Higher-order methods can be used to increase the accuracy and utilize a longer history.

Next we derive a declarative matrix representation of the above formulas. Representing numerical integration and numerical differentiation using matrices will allow us to define HCOL, which represents the necessary computations in a concise operator-based formalism (with matrices used to represent linear and bilinear operators). This opens the door to build a code/proof co-synthesis system using rewriting rules derived from mathematical (operator) identities. We define I_n as the $n \times n$ identity matrix. $[.|\dots|.]$ denotes horizontal stacking of matrices and $\oplus$ denotes the direct sum of vectors. Then,

$$\mathbf{x}^{t+h} \approx \left[\,\mathrm{I}_3\,|h\,\mathrm{I}_3\,\right](\mathbf{x}^t \oplus \mathbf{v}^{t+h}) \quad \text{and} \quad \mathbf{v}^{t+h} \approx 1/h\left[\,\mathrm{I}_3\,|-\mathrm{I}_3\,\right](\mathbf{x}^{t+h} \oplus \mathbf{x}^t).$$

We now can define a matrix that computes from the state at t and the measurements at $t+h$ the disagreement between the first-order model and the measurements. The result is a residue vector $\mathbf{r}^{t+h} = (r_x^{t+h}, r_y^{t+h}, r_z^{t+h}, r_{v_x}^{t+h}, r_{v_y}^{t+h}, r_{v_z}^{t+h})$ with $r_x^{t+h} = x^{t+h} - (x^t + hv_x^{t+1})$ and $r_{v_x}^{t+h} \approx v_x^{t+h} - (x^{t+h} - x^t)/h$. In block-matrix notation the residue vector becomes

$$\mathbf{r}^{t+h} = \mathbf{R} \cdot (\mathbf{x}^t \oplus \mathbf{v}^t \oplus \mathbf{x}^{t+h} \oplus \mathbf{v}^{t+h}) \quad \text{with} \quad \mathbf{R} = \begin{bmatrix} -\mathrm{I}_3 & -h\,\mathrm{I}_3 & \mathrm{I}_3 & 0_3 \\ 1/h\,\mathrm{I}_3 & 0_3 & -1/h\,\mathrm{I}_3 & \mathrm{I}_3 \end{bmatrix}. \tag{1}$$

Finally, we compute the L_2 norm of the residue vector, $e^{t+h} = \|\mathbf{r}^{t+h}\|_2 = (\mathbf{r}^{t+h})^\top \mathbf{r}^{t+h}$. Defining the state at time t as $\mathbf{s}^t = (\mathbf{x}^t \oplus \mathbf{v}^t)$ we can define the first order error operator

$$\mathrm{E}_h : (\mathbf{s}^t, \mathbf{s}^{t+h}) \mapsto (\mathbf{s}^t \oplus \mathbf{s}^{t+h})^\top \left(\mathbf{R}^\top \mathbf{R}\right) (\mathbf{s}^t \oplus \mathbf{s}^{t+h}). \tag{2}$$

*http://www.blackirobotics.com/

In this case the error operator can declaratively be represented as a matrix representing a quadratic form (a bilinear operator) and can be evaluated solely with matrix-vector operations. The error operator (2) enables a consistency check between the real location and velocity sensor-based estimates and the respective real and estimated velocities. An exception is detected if the error value crosses a threshold that makes it unlikely for the deviation to be caused by mere noise, but indicates a fault or attack.

The error residual consistency check that we show here is much simplified but illustrates the basic idea behind the more general hypothesis-testing driven approach that High Assurance SPIRAL uses for abnormality and attack detection. Note that in this example we trust the previous data point as soon as it is accepted (no exception is raised). This way there is no accumulation of discrepancy between measurements and model. The matrix representation of the test statistic (the residual) above continues to hold in general practical scenarios involving batch or sequentially obtained sensing data, where instead of considering only a single system snapshot one incorporates longer time-series histories and prior data for more accurate and reliable failure detection.

High Assurance Fail-Safe Controller Expression in HCOL. Based on the consistency check, we now discuss a high assurance cruise control system. In normal mode it controls the speed with a proportional-integral-derivative (PID) controller. When receiving a failure/attack signal from the virtual high assurance velocity sensor (i.e., location predicted from old position and current velocity strongly disagrees with measured current position) the controller switches to failsafe mode and brings the vehicle to a controlled stop.

We define the current velocity at time t as $\mathbf{v}^t$ and the set point velocity as $\mathbf{v}^0$. The error at time t is the difference between these velocities, $\mathbf{e}^t = \mathbf{v}^0 - \mathbf{v}^t$. Assuming the controller performed n time steps of length h, a first order discretized PID control stimulus $\mathbf{u}^t$ at time point t is computed by

$$\mathbf{u}^t = k_p \mathbf{e}^t + k_i \sum_{i=0}^{n-1} \mathbf{e}^{ih} + k_d \frac{\mathbf{e}^t - \mathbf{e}^{t-h}}{h} \tag{3}$$

with the three gain parameters k_p, k_i, and k_d. We define a history vector $\mathbf{s}^t = \mathbf{e}^t \oplus \sum_{i=0}^{n-1} \mathbf{e}^{ih}$. Then we can compute the control stimulus $\mathbf{u}^t$ and update the the history vector as function of new measurement $\mathbf{v}^t$ and the state from one time step before ($\mathbf{s}^{t-h}$) using the following matrix equation:

$$\begin{pmatrix} \mathbf{u}^t \\ \mathbf{s}^t \end{pmatrix} = \left(\left[k_p \, \mathrm{I}_e \,|k_i \, \mathrm{I}_3 \,|k_d/h \, \mathrm{I}_3 \, \right] \begin{bmatrix} \mathrm{I}_3 & \cdot & \cdot \\ \mathrm{I}_3 & \cdot & \mathrm{I}_3 \\ \mathrm{I}_3 & -\mathrm{I}_3 & \cdot \end{bmatrix} \begin{bmatrix} \mathrm{I}_3 & -\mathrm{I}_3 & \cdot \\ \cdot & \cdot & \mathrm{I}_6 \end{bmatrix} \right) \oplus \begin{bmatrix} \mathrm{I}_3 & -\mathrm{I}_3 & \cdot \\ \mathrm{I}_3 & -\mathrm{I}_3 & \mathrm{I}_3 \end{bmatrix} (\mathbf{v}_0 \oplus \mathbf{v}^t \oplus \mathbf{s}^{t-h}). \tag{4}$$

The failsafe mode of the controller is simply slowing down the car at a constant rate by applying constant brake pressure, denoted by $\mathbf{u^t} = -c$. The failsafe controller illustrates an important point. It introduces switching into the dynamics, which, in real systems, happens for various reasons and turns the model into a hybrid model. What is important in our context is that switching is outside the matrix abstraction of linear transformations. Instead, we use an operator abstraction that naturally extends the matrix abstraction (matrices represent linear or quadratic operators). We define a conditional operator

$$\mathrm{COND}_c(\mathbf{A}, \mathbf{B}) : x \mapsto \begin{cases} \mathbf{A}(x) & c \text{ true} \\ \mathbf{B}(x) & c \text{ false} \end{cases}$$

that is a predicated operation akin to the C construct `?:`. if condition c (parameterizing the operator) is true, operator $\mathbf{A}$ gets used, otherwise operator $\mathbf{B}$. The result is a conditional operator that permits state and control flow, but in a reasonably limited form. Using the conditional operator, the residue operator and the PID operator we can formally describe a fail-safe high assurance cruise control system.

3.2 Co-Generating Proof and Code for a Controller Specification

We continue our example to show how the structured matrix-vector products (that arise as in the previous section) are translated into source code in a two-step process. In a first step the matrices are decomposed into iterative sums of matrices, and then we give rules how to translate every construct into a code piece. The result is code and a complete "paper trail" of rule applications that are mathematical identities and have been used internally to translate the specification into the code, and thus provide a proof that the specification and the code are equivalent.

Formal code synthesis. Below, we denote the canonical n-dimensional row base vector (all elements zero except the ith element) with $e_i^{1\times n} = [0, \ldots, 0, 1, 0, \ldots, 0] \in \mathbb{R}^{1\times n}$ and the canonical n-dimensional column base vector with $e^{n\times 1} = (e_i^{1\times n})^\top$. The identity matrix I_n can be decomposed into a sum of n rank-1 matrices:

$$I_n \rightarrow \sum_{i=0}^{n-1} e_i^n \, I_1 (e_i^n)^\top = \sum_{i=0}^{n-1} S_i \, A_i \, G_i \tag{5}$$

which follows a general gather-compute-scatter pattern where the iterative sum contains a product of three matrices (a parameterized gather matrix G_i, a computational kernel A_i and a parameterized scatter matrix S_i). Note that the mathematical identity is written using $\rightarrow$ which introduces a direction and turns the identity into a rule, because the matrix expression on the right is a step towards the final synthesized code.

In this representation, the iterative sum is a high-level representation for a loop and gather/scatter operations represent array accesses while the kernel represents the actual computation. The following example is another optimization rule (again a matrix identity with "direction" capturing a loop merging optimization):

$$\left[\sum_{i=0}^{n-1} S_i \, A_i \, G_i \,\middle|\, \sum_{i=0}^{n-1} S_i \, B_i \, G_i\right] \rightarrow \sum_{i=0}^{n-1} S_i \left(\big[A_i | B_i\big] \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) G_i \, .$$

Applying such rules recursively expands a specification into a high-level representation of the loop-based program that implements the specification. The final result is an expression that can be evaluated into a matrix and is equivalent to the specification but at the same time represents a program. The rule system is implemented in a computer algebra system that supports symbolic (infinite precision) computation. This allows at any point to compare (sub)expressions to each other and to specifications in an mathematically exact way and is a core contributor to code/proof co-synthesis. A DSL that implements the mathematical objects and rules can be modeled after SPIRAL's SPL and OL, and has the flavor of LISP or Scheme.

Code generation. The sum-based matrix representation is translated into code using a rule set similar to the one below.

$$\begin{aligned}
\text{Code}\big(y = (A\,B)x\big) \quad &\rightarrow \quad \big\{\texttt{Decl}(t), \text{Code}\big(t = Bx\big), \text{Code}\big(y = At\big)\big\} \\
\text{Code}\left(y = \left(\sum_{i=0}^{n-1} A_i\right) x\right) \quad &\rightarrow \quad \big\{y := \vec{0}, \texttt{for}(i = 0..n-1)\ \text{Code}\big(y{+} = A_i x\big)\big\} \\
\text{Code}\big(y = e_i^{1\times n}\, x\big) \quad &\rightarrow \quad y[0] := x[i] \\
\text{Code}\big(y = e_i^{n\times 1}\, x\big) \quad &\rightarrow \quad \big\{y = \vec{0}, y[i] := x[0]\big\}
\end{aligned}$$

Note that for many cases we can prove that we do not need to initialize the vector y in the for loop and that the accumulate operation becomes an assignment.

We show as example the code generated for $\mathbf{y}^{t+h} = \big[\, I_3 \,|h\, I_3 \,\big](\mathbf{x}^t \oplus \mathbf{v}^{t+h})$ with h being a code synthesis time parameter.

```
let(y:=var(TArray(TReal, 3)), xv:=var(TArray(TReal, 6)), h := TReal(1/100),
  func([inparam(xv), outparam(y)],
    loop(i, [0..3], chain(
      assign(nth(y, i), add(nth(xv, i), mul(h, nth(xv, add(i,3)))))))))
```

This representation is a data structure in the underlying computer algebra system that executes the rule system. The code objects support symbolic execution, which allows to prove that the program is equivalent to the higher-level representations and the specification. It also allows us to build an rule-based optimizing compiler that performs standard backend optimizations like array scalarization, constant folding, and common subexpression elimination.

In a last step the code is unparsed into a restricted dialect of C and passed to a verified/certified compiler. The binary then can be empirically tested for numerical equivalence to the various levels of symbolic/fully accurate representations, including the specification. The synthesized code is highly optimized both structurally (loop optimizations) and inside basic blocks (all the standard compiler optimizations), as the high level intermediate language and the original specification enables optimizations due to better information of code semantics. Parallelization, vectorization, tiling, and similar

optimizations become much easier than in a general compiler setting. This also allows us to use a rather unoptimizing verified compiler like CompCert, because the optimizations have already been done in the synthesizer. Below we show simple ANSI C code for our example, the 3D Euler step.

```
void euler(int  *Y, double  *X, double h) {
    double q1, q2, q3, q4, q5, q6;
    q1 = X[0];
    q2 = X[1];
    q3 = X[2];
    q4 = (X[3]*h);
    q5 = (X[4]*h);
    q6 = (X[5]*h);
    Y[0] = (q1 + q4);
    Y[1] = (q2 + q5);
    Y[2] = (q3 + q6);
}
```

Proof generation. The co-generation of proofs together with the synthesis of an implementation is a core component of our system. Our system provides a framework to verify various types of rules needed, and the verification of rule application during a rewriting process. Together this provides a gap-free chain of evidence (a proof) that the produced code is equivalent to the specification. Specifically, our rule-based proof generation requires: 1) Translation ruled from specification into highest-level internal representation (matrix and operator expressions, HCOL). 2) Transformation rules inside HCOL. 3) Lowering to iterative matrix/operator expressions (lower level HCOL variant called Sigma-HCOL). 4) Transformation rules inside Sigma-HCOL. 5) Lowering to internal code representation. 6) Transformation rules for the internal code representation. 7) Output of internal code as restricted C code. 8) Symbolic execution of generated C programs to recover the semantics and map it back to the internal representation. All rules are implemented as rewriting rules with pattern matching inside the computer algebra infrastructure. For instance, rule (5) is shown in its source code representation below. `@` is the pattern matching wildcard and the expression matches `I(n)` for any `n`. On the right-hand side of the rule, `@` refers to the matched object (`I(n)`) and `@1` to is parameter (`n`).

```
SumSAG_In := Rule(@(I(@1)),
   (@, @1)->Let(i := Idx(@1), ISum(i, @1, e(@1, i) * I(1) * e(@1, i)^T)));
```

A rewriting trail used to construct a proof shows the matched patterns and how the rule replaced the matched sub-expression. For instance, applying the rewrite rule `SumSAG_In` to an instantiation of `residue` by calling `RewriteOnce(residue(1/100), SumSAG_In);` leads to the following output:

```
> rule: "SumSAG_In"
> matched: BlockMat([[-I(3), 1/100*I(3), I(3), O(3)],
>             [100*I(3), O(3), 100*I(3), @(I(3))]])
> wildcards: @="I(3)", @1="3"
> rewritten: "ISum(k, 3, e(3, k) * I(1) * e(3, k)^T))"
> proof: "I(3) == ISum(k, 3, e(3, k) * I(1) * e(3, k)^T))"
> result: BlockMat([[-I(3), 1/100*I(3), I(3), O(3)],
>            [100*I(3), O(3), 100*I(3), ISum(k, 3, e(3, k)*I(1)*e(3, k)^T)]])
```

This output shows the details of the matching/substitution operation and can be translated into a proof that verifies the assertion labeled `proof`.

The final proof is performed using the proof assitent Isabelle[†]. The trail of rewriting operation then is turned into an Isabelle proof script (ISAR), and rewrite rules become Isabelle lemmas. The combination of lemmas and an ISAR script provide enough hints to Isabelle to enable automatic proofs to go through, showing that the generated code is indeed equivalent to its declarative specification.

[†] `https://www.cl.cam.ac.uk/research/hvg/Isabelle/`

4. CONCLUSION

This paper provides a first glimpse at High Assurance SPIRAL, an extension to the SPIRAL system aimed at co-synthesizing highly efficient code and proofs for control algorithms. The system's input is a high-level declarative specification of control algorithms and the output is 1) a highly optimized sound implementation of the algorithm, and 2) a formal proof that the implementation is indeed implementing the specification faithfully. The aim of the tool is to close the gap between formal methods tools that prove the correctness of control algorithms and their numerical implementation using modern computer systems with performance-enhancing instruction sets. A crucial component of our approach is to leverage domain knowledge to enable reasoning about otherwise hard to prove properties and implementation issues like floating-point arithmetic.

5. ACKNOWLEDGEMENT

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0291. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] Moura, J. M. F. and Püschel, M., "SPIRAL: An Overview," in [*Proc. Workshop on Optimizations for DSP and Embedded Systems*], (2003). Held in conjunction with CGO.
[2] Püschel, M., Singer, B., Veloso, M., and Moura, J. M. F., "Fast automatic generation of DSP algorithms," in [*Proc. Int'l Conf. Computational Science (ICCS)*], *LNCS* **2073**, 97–106, Springer (2001).
[3] Püschel, M., Singer, B., Xiong, J., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., and Johnson, R. W., "SPIRAL: A generator for platform-adapted libraries of signal processing algorithms," *Int'l Journal of High Performance Computing Applications* **18**(1), 21–45 (2004).
[4] Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N., "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* **93**(2), 232– 275 (2005).
[5] Franchetti, F., Püschel, M., Voronenko, Y., Chellappa, S., and Moura, J. M. F., "Discrete Fourier transform on multicore," *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"* **26**(6), 90–102 (2009).
[6] Püschel, M. and Moura, J. M. F., "The algebraic approach to the discrete cosine and sine transforms and their fast algorithms," *SIAM Journal of Computing* **32**(5), 1280–1316 (2003).
[7] Püschel, M. and Moura, J. M. F., "Algebraic signal processing theory: Foundation and 1-D time," *IEEE Trans. on Signal Proc.* **56**(8), 3572–3585 (2008).
[8] Püschel, M. and Moura, J. M. F., "Algebraic signal processing theory: 1-D space," *IEEE Trans. on Signal Proc.* **56**(8), 3586–3599 (2008).
[9] Püschel, M. and Moura, J. M. F., "Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs," *IEEE Trans. on Signal Proc.* **56**(4), 1502–1521 (2008).
[10] Solar-Lezama, A., Rabbah, R., Bodík, R., and Ebcioğlu, K., "Programming by sketching for bit-streaming programs," *SIGPLAN Not.* **40**, 281–294 (June 2005).
[11] Vechev, M., Yahav, E., and Yorsh, G., "Inferring synchronization under limited observability," in [*Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*], *TACAS '09*, 139–154, Springer-Verlag, Berlin, Heidelberg (2009).
[12] Frigo, M. and Johnson, S. G., "The design and implementation of FFTW3," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* **93**(2), 216–231 (2005).
[13] Whaley, R. C. and Dongarra, J., "Automatically Tuned Linear Algebra Software (ATLAS)," in [*Proc. Supercomputing*], (1998). `math-atlas.sourceforge.net`.
[14] Im, E.-J., Yelick, K., and Vuduc, R., "Sparsity: Optimization framework for sparse matrix kernels," *Int'l J. High Performance Computing Applications* **18**(1) (2004).
[15] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Orti, E., and van de Geijn, R., "The science of deriving dense linear algebra algorithms," *TOMS* **31**, 1–26 (March 2005).