AUTOCODING CONTROL SOFTWARE WITH PROOFS I: ANNOTATION TRANSLATION

Romain Jobredeaux, Timothy E. Wang, Eric M. Feron
Georgia Institute of Technology, Atlanta, GA

Abstract

In an effort to meet the reliability standards that control systems operating in safety-critical roles require, we have started laying the foundation for a tool-set that migrates control theory properties and proofs into the software implementation of those control systems designs. By using this tool the engineer can provide a more rigorous guarantee of the quality of the software and initiate the formal verification process. The tool focuses on control software in order to leverage the domain knowledge from existing mathematical techniques for the analysis and synthesis of control systems. As a first step in the development of the tool-set, we have created a prototype of a Scilab to C translator with proof annotation support. Though limited in its current functionalities, the development of this prototype allowed us to identify the key issues which will be used to further refine the translator. This paper describes the prototype and the further improvements planned for the translator.

Introduction

Safety-critical applications have become an important part of modern life. One can find many examples of safety-critical applications in the fields of aerospace, defense, medical, nuclear and mass transportation. Whether it is the avionics in a commercial passenger jet, the heart valve inside a cardiovascular impaired patient, or the automatic speed and signaling system in a high-speed railway network, one thing they all have in common is the presence of embedded control systems. As the level of automation increases in the modern world, one can expect more and more embedded control systems that operate in safety-critical roles. The notion of what is safety-critical refers to the unacceptable characteristic of the consequence due to failure. The consequence can be the loss of human life, massive economic and property damage, or severe disruptions to the environment. The reliability of these embedded control systems therefore is very important.

An embedded control system has both hardware and software components. To ensure the quality of the software component, it is usually submitted through a rigorous certification process that involves both extensive testing and analysis using mathematics-based verification techniques developed by the formal methods community. Sophisticated automated or semi-automated tools already exist and are being used by the industry for the formal analysis of complex computer programs. Despite these advances, one limitation in program analysis remains in that the properties from the program specifications can be arbitrarily complex and proving them can be difficult even by hand, let alone computers. This limitation comes from an inherent underlying undecidability issue. Fortunately it is common practice in the control engineering community to provide rigorous safety analysis of control designs before their implementation. For example take the following linear compensator in the state-space form.

$$x_c^+ = A_c x_c + B_c y \qquad (1)$$
$$u = C_c x_c + D_c y$$

Given the input y that is bounded, A_c being a Hurwitz matrix, (A_c, B_c) is controllable, we can prove by hand that the control system in (1) is invariant with respect to an ellipsoid parameterized by a matrix P: $\mathcal{E}_P \overset{\Delta}{=} \{x \in \mathbb{R}^n | x^T P x \leq 1\}$ where P is a solution to a linear matrix inequality (LMI) [1]. As proposed and demonstrated in [2], this property along with the proof if inserted into the code in a format such as Hoare triples [3] can be helpful to the program analyzer in formally verifying the code.

One of the obstacles to using control theory to assist the formal analysis of control software is that two communities effectively operate in different worlds and speak different technical languages. For example the control engineer is comfortable with state-space systems while the formal method scientist deals with finite-state machines. Also the fact remains that for all but the simplest examples, manually transforming and inserting control theoratic properties

978-1-61284-798-6/11/$26.00 ©2011 IEEE

and proof into line by line assertions about the output code is prohibitively time consuming. One solution is to have a tool-set that can translate the complete control system specifications into annotated code in an automatic fashion. For that purpose we propose a tool-set coined informally as an *autocoder with proofs* that satisfies the following general requirements. 1. The front-end environment for the creation of the control system specifications should be familiar to the control engineers. 2. The front-end specification language should have precise semantics for both controller design and property/proof annotations. 3. The back-end code generator can translate both the controller specifications into code as well as the properties and proofs into code annotations. 4. The generator's output is analyzable by tools developed by the formal methods community.

The organization of the remainder of this paper is as follows. First, a short overview of the proposed tool-set is described. From there we give the reasons for the development of the Scilab to C translator, then a technical description of the Scilab to C translator is given. In the ensuing sections a description of its shortcomings are provided. Afterwards we recommend a set of refinements for the translator and as well as new functionalities and specifications for the next prototype translator. Finally a discussion on future improvements on the overall tool-set is provided.

A Description of the Tool-Set

Front-End

As described in more detail in [4] the front-end environment is Simulink-like i.e. a data-flow language underlying a graphical interface featuring blocks, subsystems, and signal wires. Simulink being the industry's standard platform, is the natural choice due to its familiarity to control engineers. The front-end environment is further enhanced with the semantics of control proofs and properties. This allows the insertions of those properties and proofs at the control design level. A graphical example of this functionality is shown in Figure 1 using existing Simulink blocks.

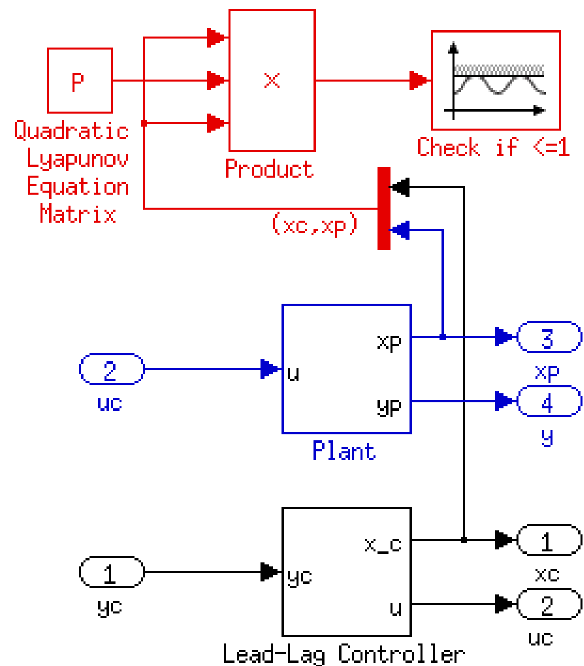

Figure 1. Graphical Annotation of the Control System

In this example, the full control system specifications is consisted of a lead-lag controller, the plant model, a closed-loop Lyapunov function parameterized by P, and the invariant property: $(xc, xp)^T P(xc, xp) \leq 1$. The invariant property is inserted into the Simulink diagram as the annotation. This can be done graphically by connecting the Lyapunov function blocks with the signals that represent the controller and plant states. The value of P can be generated automatically by a robust control analysis toolbox such as IQC-β[5] or μ-tools[6].

Back-End

The back-end of the tool-set is a translator that transforms the control system specifications into the annotated program in an industrial language. This necessity for an *autocoder with proofs* led to the development of the prototype translator described in this paper. The target annotation language for the translator can be a matter of preference, and is also dependent on the target language. For the demonstration of the prototype translator in this paper, we picked the target language based on the criteria of industry popularity and the amount of support in terms of program analysis tools by the formal methods

community. We settled on the C language, which is commonly used in the industry. Naturally we also decided to go with the ANSI C Specification Language (ACSL) [7] for the annotations. Supporting ACSL are a rich set of program analyzers such as Frama-C with its jessie and weakest precondition (wp) plugin, [8], which can generate the necessary proof conditions to be discharged by a variety of interactive theorem provers such as the Prototype Verification System by SRI [9].

The choice of the source language logically would be the data-flow language used in the front-end environment. However due to the multitude of synchronous data-flow languages used in the industry, we decided,for the first prototype translator, to use a source language that was easy to annotate and transform. We eventually settled on the translation layers as shown in Figure 2. The Simulink language cannot be altered so easily and doesn't have native commenting capability. Simulink's autocoder Real-Time Workshop uses an intermediate language for translation that is internal and also unavailable for alteration. Simulink is however part of Matlab. One of the advantages of this integration with Matlab is the ability for the user to insert any Matlab code into a Simulink diagram. The entire Simulink design can be one Matlab function.

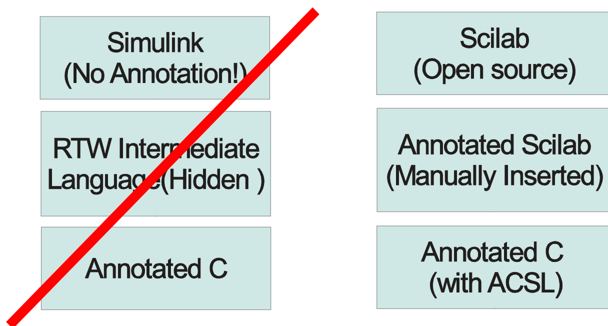

Figure 2. Language Layers

Matlab itself is a popular language among control engineers for doing design and simulation. The syntax and matrix manipulation capabilities of Matlab allows for compact description of control properties and proofs. We can manually, and with ease provide these proofs and properties as Matlab comments. Matlab's imperative style also makes it relatively easy to translate to C. These conveniences coupled with the fact that there is an open-source version of Matlab called Scilab led to the final decision to choose Scilab as the source language in our prototype *autocoder with proofs*. Scilab also has a Simulink-like modeling environment called Scios which can be adapted to be the front-end layer. However for this early version of our prototype translator, we are looking to demonstrate the translation of control annotations rather than the best way for the control engineer to provide those annotations.

Scilab to C Translator

Scilab is an open source numerical computation software created by the French National Institute for Computer Science and Control (INRIA). It is a project that is currently backed by an international consortium of 25 members from both the industry and academia. For the rest of this paper we shall refer to Scilab as both the software package and the technical computing language.

Scilab Semantics

Scilab is a high-level interpreted, imperative language with similar array programming capabilities as Matlab. Like Matlab, the type system in Scilab was designed mainly with convenience of scientific computing in mind. The main data type is the matrix which is expressed in Scilab as

$$[x_{11},\dots,x_{1m};x_{21},\dots,x_{2m};\dots x_{n1},\dots,x_{nm}] \quad (2)$$

where the entries x_{ij} can be either floats or booleans. There are other data types in Scilab but we only consider the following list for the translator (see Figure 3). Let $\mathbb{F}$ be the quasi-ring of floating-point numbers.

$$\begin{aligned} &Scalar: \mathbb{F} \\ &Matrix: \mathbb{F}^{n\times m} \quad (3) \\ &Boolean: \{0,1\} \\ &BooleanMatrix\{0,1\}^{n\times m} \end{aligned}$$

Figure 3. Basic Data Types in Scilab

The conversion from the one element matrix i.e. $\mathbb{F}^{1\times 1}$ to the scalar type is implicit. To the translator, all matrices with one element are considered as scalars. The vector x_c from the state-space representation of the linear controller in (2) is expressed as a matrix of dimension $n\times 1$.

In the translator we consider the following set of operations (see Figure 4). The scalar operations plus,

minus, multiply and divide $\{+,-,.*,./\}$ are generalized to operate element-wise on matrices. In addition, basic linear-algebra operations such as matrix multiplication and matrix transpose $\{*,'\}$ are also supported by Scilab. Their semantics are formalized internally and presumably in a more strongly-typed language. For the formal verification of these operations, efforts such as the one in [7] to formalize linear algebra semantics in a theorem prover has led to a linear algebra module in the NASA langley PVS libraries. The mixed matrix scalar binary operations $\{+,*,/\}$ are also considered. The necessity comes from the need for expressing certain proof formulas that is the end product of a relaxation procedure using Lagrange multipliers, see S-Procedure in [11]. The $\{-\}$ operation is skipped over because it is not needed for expressing any of the control semantics that we have mentioned.

$$MatrixAdd/Subtract\{+,-\}: \mathbb{F}^{n\times m} \times \mathbb{F}^{n\times m} \to \mathbb{F}^{n\times m}$$

$$MatrixElementOp: \{.*,./\}: \mathbb{F}^{n\times m} \times \mathbb{F}^{n\times m} \to \mathbb{F}^{n\times m}$$

$$MatrixMultiplication: *: \mathbb{F}^{n\times m} \times \mathbb{F}^{m\times k} \to \mathbb{F}^{n\times k}$$

$$MatrixTranspose': \mathbb{F}^{n\times m} \to \mathbb{F}^{m\times n}$$

$$\boldsymbol{Matrix\&ScalarOp}: \{+,*,/\}: \mathbb{F}^{n\times m} \times \mathbb{F} \to \mathbb{F}^{n\times m} \quad (4)$$

Figure 4. Basic Math Operations in Scilab

In addition to operations we also translate the following binary relations in Figure 5. The binary relations on scalars are used both in the code and the annotations.

$$\diamond \in \{<,>,<=,>=,==,<>\} \quad (5)$$

$$ScalarRelations: \diamond: \mathbb{F} \times \mathbb{F} \to \{0,1\}$$
$$MatrixRelations: \diamond: \mathbb{F}^{n\times m} \times \mathbb{F}^{n\times m} \to \{0,1\}^{n\times m}$$

$$\boldsymbol{Matrix\&ScalarRelations}: \diamond: \mathbb{F}^{n\times m} \times \mathbb{F} \to \{\mathbf{0},\mathbf{1}\}^{n\times m} \quad (6)$$

Figure 5. Basic Relations in Scilab

Scilab support commenting in the code using the symbol "//". We specified a new commenting symbol "//@" to denote annotations that are to be translated instead of being discarded. For example the loop invariant $x \in \mathcal{E}_P$ is expressed in Scilab as in Figure 6.

```
1 //@ transpose(x_0)*inverse(P)*x_0<=1
```

Figure 6. Scilab Annotation

The flow control in Scilab shares many common elements with C. Both languages have the *while* and *for* loops as well as the *if-else-then* statement. One key difference is the way that the *for* loop variable is handled. In Scilab the loop variable is initiated to a matrix. We denote this matrix the *loop matrix*. The total number of loop iterations corresponds to the number of columns in the *loop matrix*. Before the nth iteration, the nth column of the *loop matrix* is assigned to the loop variable implictly by the Scilab interpreter. We provide the following Scilab code (see Figure 7) and the C code output (see Figure 8) to highlight this difference.

```
1 for (i = [1,2;3,4])
2 j = i+1
3 end
```

Figure 7. *for* loop in Scilab

The output C code in Figure 8 has an index variable *i_0_loopIndx* that keep track of the current iteration number and the variable *i_0_loopMat* to hold the values of the *loop matrix*. In addition the function ***submatrix*** is used to return a column of the *loop matrix*.

```
1  #include "tools.h"
2
3  int main(int argc, char *argv[])
4  {
5    char false;
6    double** i_0;
7    int i_0_loopIndx;
8    double** i_0_loopMat;
9    double** j_0;
10   char true;
11   i_0 = allocM ( 2, 1);
12   i_0_loopMat = allocM ( 2, 2);
13   j_0 = allocM ( 2, 1);
14   i_0_loopMat[0][0]=1;
15   i_0_loopMat[1][0]=3;
16   i_0_loopMat[0][1]=2;
17   i_0_loopMat[1][1]=4;
18   for ( i_0_loopIndx=0 ; i_0_loopIndx < 2 ;
        i_0_loopIndx++ )
19  {
20     i_0 = submatrix (i_0_loopMat,0,i_0_loopIndx, 2, 1);
21     j_0 = add_scalar(1, i_0,2,1,0);
22
23  }
24  return 0;
25
26 }
```

Figure 8. C Code Output

Description of ACSL and Frama-C

ANSI C Specification Language (ACSL) is a specification language developed by the French Atomic Energy Commission (CEA) and INRIA. Using ACSL we can insert annotations expressing the expected functional and safety properties of C functions. The annotation format is in the style of a Hoare triple. Given a function call <myfunction> we can label a contract on the function that is consisted of a required pre-condition P that guarantees a certain post-condition Q after the function call is executed.

$$\{P\} < myfunction > \{Q\} \quad (7)$$

The pre and post-conditions are first-order logic formulas. The syntax of the ACSL contract is shown in Figure 9. The pre-condition is indicated by the keyword *requires* and the post-condition is indicated by the keyword *ensures*.

```
1 /*@ requires <pre-condition>
2   @ ensures <post-condition>
3   */
4 <type> myfunction(<arguments list>)
```

Figure 9. Example of ACSL Contract

For memory safety we can specify pre-conditions on pointer validity using the keyword *valid*. For example to check the validity of an array of size 2 i.e. A[2] we can write the following pre-condition:

```
1 /*@ requires (A+(0..1)) \valid (A + (0..1))
```

Frama-C is a set of tools also developed by the CEA and INRIA for the purpose analyzing C programs. The jessie plug-in in Frama-C can be used to generate verification conditions from the ACSL annotated code, and then outputs the verification conditions to interactive and automated theorem provers. It generates verification conditions by computing the weakest pre-condition that can guarantee the post-condition. For example, if we have an ACSL contract that requires P and ensures Q for the function call S, and given that the P' is the weakest precondition for Q and S i.e. $P' = wp(S, Q)$ then the verification condition generated by jessie is

$$P \Rightarrow P' \quad (8)$$

Translation Scheme and Verifying C Functions

The approach that we have taken in the translation process is tailored so the output will be suitable for analysis by Frama-C's jessie plugin. We translate only the subset of Scilab that is used to implement linear control programs as well as expressing Lyapunov stability proofs, S-procedure and ellipsoids. The following points describe the the translation scheme and some of the formal verifications done on the C functions.

I. The matrix type is translated into double pointers of the double type. We also implemented a memory allocator function allocM to dynamically [1] create the 2-dimensional array of the correct size for storing the values from the matrix.

II. The Scilab operations are implemented into C functions using the formal semantics described earlier. The translator then maps the operations in the Scilab code into their corresponding C functions.

III. The binary relations for matrices are also implemented as C functions but the details are skipped since they are not relevant to controller implementation or annotation.

IV. The C functions that implement the Scilab operations are verified using jessie and ACSL for their functional correctness. Specifically the semantics of the C functions are verified against the formalized semantics of the Scilab operators.

V. No control semantics i.e. $x \in \mathcal{E}_P$ are verified for the C functions namely because without knowing their point of call in the overall code, no pre-conditions from control system analysis can be inserted.

VI. The language of annotation is also in Scilab. The annotation translation is handled exactly the same way as the code i.e. using the C functions to replace Scilab operators.

[1] This dynamic allocation is done only once at the beginning of the program to initiate the array.

VII. The prototype Scilab to C translator can be found in Table 1. Here in the appendix we provide an example of annotated Scilab code and the translated output.

VIII. Strictly speaking, the annotated C code output is not directly analyzable by Frama-C with jessie. A preprocessor is needed to format the pre-condition and post-conditions as Frama-C contracts as well as convert each line of code to a function call.

Table 1. Scilab Operations Mapped to C Functions

$\{+,-,.*,./\}$	add, substract, mult_elem, div_elm
$\{*,'\}$	mult, transpose
$\{+,*,/\}$	add_scalar, mult_scalar, div_scalar

Discussion

The motivation behind implementing the translator to directly map Scilab operators to verified C functions can be summarized as follows. Using these C functions we can generate C code that is similar to the original Scilab code in form. This improves the readability of the annotations as well as the code. Furthermore the translator does not have to perform ellipsoid calculus to generate the ellipsoid invariants for all the intermediate steps in a matrix operation. The control semantics can then be verified essentially on the Scilab level and the result can be assumed to hold true for the output C code with all the function calls inlined. The reasoning behind this claim is as follows: since all the translator has done is to directly copy over the C functions to replace the Scilab operators, and since the semantics of these C functions are already verified against the semantics of the Scilab operators that they replaced, thus any invariant from control theory that holds true in the Scilab code should also hold for the output of our prototype translator. This argument was formed during the development of the translator. However there are several issues with it. One being that the choices made for the translation scheme which led to this line of reasoning run in counter to industry practice.

To adhere to industry practice, the translation of the code and the proof has to be as close as possible to a form that is actually executed on the target machine. This practice also fits for our purpose since we want to leave as few translation tasks as possible to the compiler, so that most of the formal verifications can be done on the source code level or above. This means all the functions should be inlined. It also means for the example linear controller in (1), an ellipsoid invariant containing all the relevant variables should be propagated into every line of the output code. Having C functions that are verified for their functional property and then inlined directly into the translated code leaves a lot of lines without an ellipsoid invariant.

One possible way around this problem that we have explored is to create a set of pre-built annotation templates and then insert them into a library of linear algebra C functions such as LAPACK. The template can be parameterized by an ellipsoid. During the inlining, the translator inserts the function with the annotation template. Here we give an example of such annotative template for a hypothetical C function *mult* that multiplies the A matrix by the controller state x_c. First we give the syntax of the template.

```
1  /*@ invariant (x,y)      R */
```

The comment has the semantics of the ellipsoid invariant

$$\left\{\begin{bmatrix} x \\ y \end{bmatrix} \middle| \begin{bmatrix} 1 & [x\ y]^T \\ \begin{bmatrix} x \\ y \end{bmatrix} & R \end{bmatrix} > 0\right\} \tag{9}$$

With $A = \begin{bmatrix} 1 & -1 \\ 0 & -2 \end{bmatrix}$, $x_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and the hypothetical C function *mult*, we have the following annotative template,

```
1
2   mult(double** A,double** xc, double** xtmp)
3   {
4      /*@ xc   &&   inverse(P) */ 5
5      xtmp[0][0]=A[0][0]*x[0]+A[0][1]*x[1];
6      /*@ (xc,xtmp[0][0]) &&
           T1*inverse (P)*transpose(T1) */
7      xtmp[1][0]=A[1][0]*x[0]+A[1][1]*x[1];
8      /*@ (xc,xtmp[0][0],xtmp[1][0]) &&
9        @ T2 *T1 *inverse(P) *transpose(T1)
             *transpose(T2) */
10    return 0;
11  }
```

$T1$ and $T2$ are the appropriate transformation matrices .

$$T1 := \begin{bmatrix} \mathrm{I}_{2X2} & \\ \mathrm{A}[0][0] & \mathrm{A}[0][1] \end{bmatrix}$$
$$T21 := \begin{bmatrix} \mathrm{I}_{3x3} & \\ \mathrm{A}[1][0] & \mathrm{A}[1][1] \end{bmatrix} \quad (10)$$

After inlining, the parameter P is substituted for the ellipsoid from the pre-condition of the function call.

The problem with this "static template" is that it only works for our hypothetical C function *mult* which has a very "restrictive" semantics of Ax_c where $A \in \mathbb{F}^{2\times2}$ and $x_c \in \mathbb{F}^{2\times1}$. The actual implemented C function *mult* is more general and computes the product of two matrices of any appropriate sizes. It uses several nested *while* loops and the total number of loop iterations in the function depends on the dimensions of the input matrices. To make the template idea above work with *mult*, we need to check if the second input matrix is the variable representing the controller state x_c. We also need a "dynamic template" that can used to annotate post-conditions for all the possible lines of executions due to loop iterations. The following is an example of a dynamic template for *mult*. The input matrices are A and B.

```
1  /*@ requires colA==rowA colA==rowB colB==1
2  @ ensures (B,xtmp(i,k)) && T(i,k)* inverse(P)*
   transpose(T(i,k)) */
```

The first line enforces the condition $A \in \mathbb{F}^{n\times n}$ and $B \in \mathbb{F}^{n\times1}$. The variable *xtmp* is the output array. The i,j,k are the loop variables. Both i and k iterate from 0 to $n-1$. The outer-loop variable j iterates from 0 to 1 so this loop can be ignored. For every loop iteration the function computes xtmp[i][j]+=A[i][k]*B[k][j]. The comment after "ensures" represents the post-condition after pth execution of *xtmp[i][j]+=A[i][k]*B[k][j]* where $p = i * n + k$. The expression T(i,k) has the following semantics

$$f1(p) := floor(p/n)$$

$$f2(p) := remainder(p/n)$$

$$Q_p := \begin{bmatrix} I_{(n+f1(p))\times(n+f1(p))} & Z_{(n+f1(p))\times1} U_{[f1(p),f2(p)]} \\ 1 & \end{bmatrix}$$

$$W_p := [I_{(n+f1(p))\times(n+f1(p))} Z_{1\times(n+f1(p))}] \quad (11)$$

$$M_p := \begin{cases} Q_p W_p & iff2(p) = 0 Q_p \\ iff2(p) \neq 0 & \end{cases}$$

$$T(i,k) = \prod_{p=i*n+k}^{0} M_p$$

where $U_{[f1(p),f2(p)]}$ denotes a matrix of size $1 \times (n + f1(p))$ with each entries of the matrix being 0 except

$$U_{[f1(p),f2(p)]}(1, f2(p)+1) = A[f1(p)][f2(p)] \quad (13)$$

Finally the expression *xtmp(i,k)* means $xtmp[0][0], xtmp[1][0], \dots, xtmp[i][0]$.

Refinements for the Prototype Translator

Based on the work described earlier in the paper, we have planned several changes for the next version of the prototype translator.

Simulink vs Lustre

There are many control design platforms used by the industry that are based on the synchronous language paradigm. MathWorks' Simulink is one of the most popular ones used by control engineers especially in the automobile field. Esterel's SCADE which is based on the language Lustre is another one. Built with safety-critical software in mind, SCADE has been used mostly in the aerospace, nuclear and high-speed railway sectors.

Simulink's popularity makes it the most logical choice for our interface layer, However for software verification purpose, Simulink presents a difficult challenge since it sacrifices some strict formalism for the purpose of improving usability to control engineers. Lustre on the other hand was designed primarily as a programming language[12] rather than a control design and simulation platform. Lustre is a synchronous language with data-flow like properties created in the 1980s by researchers from the French institution VERIMAG, The language was eventually integrated into the tool-set ESTEREL SCADE which is now widely used for the production of safety-critical embedded control software. Some of the important features in SCADE are its certified code generation tools to ADA/C, a built-in model checker, and a

wealth of static analysis tools in support. Like Simulink, Lustre has a graphical interface featuring wires and blocks. Although neither can be truly categorized as a programming language, Lustre has a set of more precisely-defined semantics as well as strong typing. This has led the industry (see Rockwell Collins in [13]) to place Lustre in the role of a "gateway language" between the interface layer used in the high-level design and the languages used in the formal methods tools. For these reasons, we think Lustre is a good intermediate language for our *autocoder with proofs*.

Translating Simulink to Lustre with Annotations

There is a lot of work in the literature concerning tools to translate Simulink into some other representation for the purpose of formal verification. As referenced earlier in [13], Rockwell Collins built a tool chain that translates from Simulink to Scade/Lustre, and finally from Lustre to a collection of model checkers and theorem provers. The tool described in [14] translates Simulink directly into the input language of a model-checker, and the work in [15] describes translation from Simulink to a hybrid automata. Finally these papers from [16], [17], describe efforts to translate a subset of Simulink into Lustre. We now refine our original approach to include Lustre as the intermediate layer between the Simulink-like interface and the C layer. First we give a short description of the Lustre language.

A Lustre program at its core is a set of functions over a set of flows. Each flow can be described as a function that maps an infinite sequence of natural numbers to an infinite set of real values or booleans. The sequence of natural numbers represents a clock. The flow is also analogous to the signal in Simulink with the exception that in Simulink the mapping is done from $\mathbb{R}^+$. From this basic difference, we can see that Lustre inherently has a discrete-time semantics. In addition to the scalar operations $\{+,-,*,/\}$ and basic control flow structures i.e. *if-else-then*, Lustre also has few special operators $\{pre, \rightarrow, when, current\}$. The *pre* operator is similar to the $\frac{1}{z}$ block in Simulink. The only difference is that it doesn't take an argument that specifies the initial state. The way this is done in Lustre is by using the → operator. Here is an example of Euler integration on the flow x with sampling time Ts in Lustre.

```
1 x=0->pre(x)+Ts*x
```

Each function in Lustre is called a node. The node is analogous to the blocks provided in Simulink. For example the step command block in Simulink can be implemented in Lustre with discrete-time semantics as the following:

```
1 node step_command(step_time: int, initial_value,
        final_value: real)
2 return (output:real)
3 var count: int;
4 let
5   count=0->(pre(count)+1);
6   if (count<step_time) then output=initial_value
         else output=final_value;
7 tel
```

The variable *counter* keep tracks of the number of time-steps that have lapsed. The *if-else-then* statement assigns the initial value to the output if the counter is less than the step time. Once the counter reaches the step time, the final value is assigned to the output.

There exists a tool called **Simulink2Lustre** developed at VERIMAG that can translate a subset of discrete-time Simulink into Lustre, This subset is shown in Figure 10. We are interested in adapting this tool for the translation of annotated Simulink to annotated Lustre.

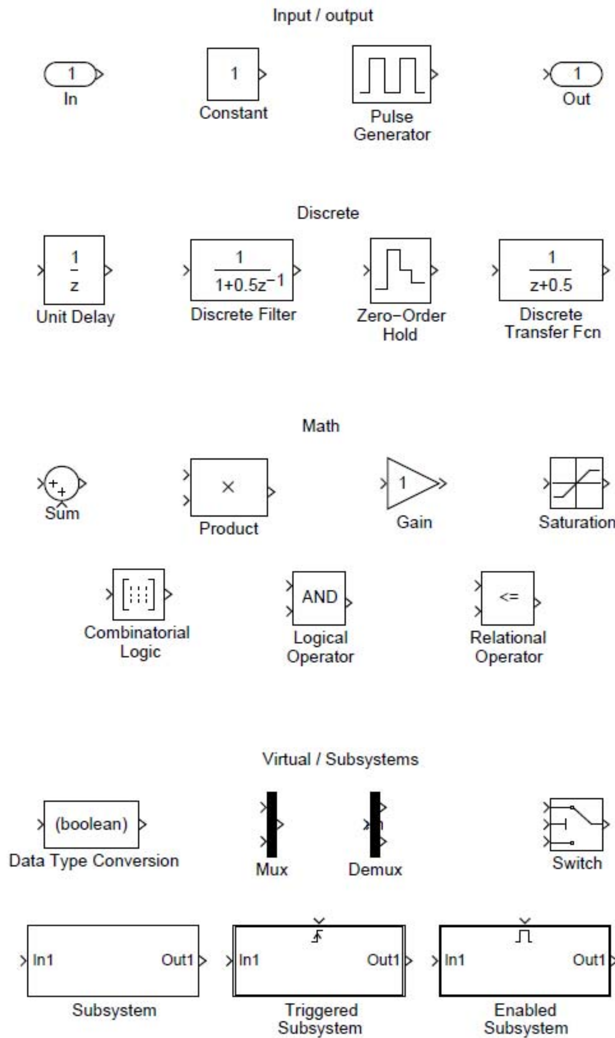

Figure 10. Translated Blocks by Simulink2Lustre [12]

Simulink2Lustre (S2L) tries to preserve the semantics of the Simulink blocks as much as possible. To verify this, the authors of the tool checked the simulation result of the output Lustre program against the simulation result of the input for a variety of Simulink models. In addition to that, our goal is also to preserve the control semantics. For example the ellipsoid invariant needs to be true in both the input Simulink model and the output Lustre program. For the most part, this can be done precisely by preserving the semantics of the Simulink blocks at least if the annotated model only uses discrete-time blocks [2]. However in adaptive control, all the high-level design and analysis is done in the continuous-time domain. The questions become how to handle the model that is constructed with continuous-time blocks. There are several issues to translating the continuous-time blocks.

I. Lustre doesn't have continuous-time semantics. Any continuous-time block in the Simulink model would have to be converted to discrete-time.

II. Lyapunov stability proof derived from the analysis of continuous-time controllers doesn't necessarily have to work for the discretized controller, although it often does.

III. Lyapunov stability proof of the discretized controller doesn't have to exist e.g. the case of adaptive controllers.

IV. A stability proof that works for one method of discretization doesn't have to work for another.

V. A discretized state-space block can have a different semantic from the discretized transfer function of the same state-space system even if the methods of discretization are the same.

VI. The semantics of the continuous-time blocks is dependent on the simulation method and the sample time.

Most of these issues are not strictly speaking a problem for the translator. For example the existence of a Lyapunov function for an Euler discretized adaptive controller is matter of control theory. To get around this problem at least for now, we can restrict the translator inputs to a class of controllers where the same stability proof exists for both the continuous-time controller and the discretized version. To accomplish this in an automatic fashion, we need to build a type checker that can analyze the Simulink model and infer the controller class.

Figuring out the set of additional Simulink blocks and types for expressing control semantics can be done in an ad-hoc manner i.e. by adding on more

[2] Actually even Simulink's discrete-time blocks have continuous-time semantics. They are actually piece-wise constant but we ignore this fine distinction for now.

features as the need to demonstrate the autocoder on more complex controllers arises in the future. For now we describe a list of blocks and types required for the case of the linear controller.

Extensions to Simulink, Lustre and S2L

For our purpose we first need some sort of annotation indicator in both Simulink and Lustre. There are two ways to do this. One is to simply add an annotation flag to the elements of both languages. Since all the linear control proofs and properties can be constructed using the existing elements in either language, indicating the annotations should be as simple as switching on a flag.

The second approach requires defining formal semantics for a new set of Simulink blocks that are specifically designed to indicate the stability property of the control system. For example the linear compensator in (1) has bounded-input and bounded output stability with a quadratic function as the certificate. In the case of Simulink, these certificate blocks also can be used to indicate that the control semantics are only valid in the field of real numbers.

Now we can describe the additional Simulink blocks that need to be translated because they are necessary for the insertion of control semantics such as Lyapunov functions, S-Procedure based inequalities, and the plant model.

I. The translator should be extended to cover continuous-time blocks as well. Indeed in nearly all control system specifications, the plant models are in continuous-time. The plant model is used in the proof of the closed-loop stability of the controller therefore it cannot be ignored. Additionally most control analysis performed at the design level are in continuous-time. To maintain the familiarity of the interface level, we need to make sure that these blocks are available to the user.

II. The semantics of a Simulink model is dependent on the method of the solver and the sampling time. To reduce the total number of possible different semantics that a model can have, we restrict the solver to Euler's method with a fixed time-step of Ts. We also fixed the sampling rate to a constant $\frac{1}{Ts}$. We can always insert higher-fidelity discretization scheme later.

III. The continuous-time transfer function and state-space blocks need to be transformed into a discrete-time form. Under the restrictions above, the only possible method of discretization is Euler's with the step-size Ts.

IV. Using Euler's discretization on the integrator block $\frac{1}{s}$ yields the transformation in Figure 11. The transformation identity used is $s = \frac{z-1}{Ts}$. We can easily see that the transformed model maps directly to Lustre operators and flows.

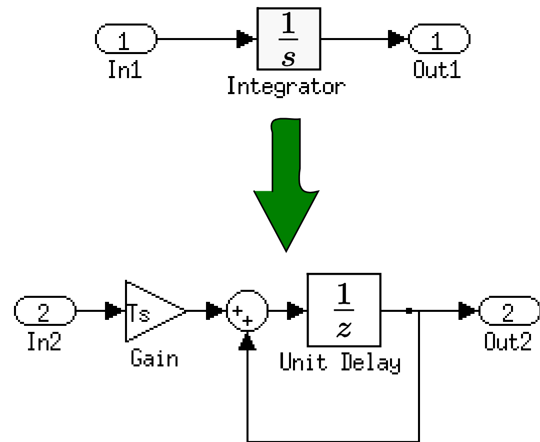

Figure 11. Continuous -Time Integrator to Discrete-Time

V. Using Euler's discretization scheme on a proper transfer function such as the one in top of Figure 12 yields a discrete-time model consisting of the following blocks: *adder*, *forward difference*, *unit delay* and *gains*, This output can also be mapped directly to Lustre operators and flows.

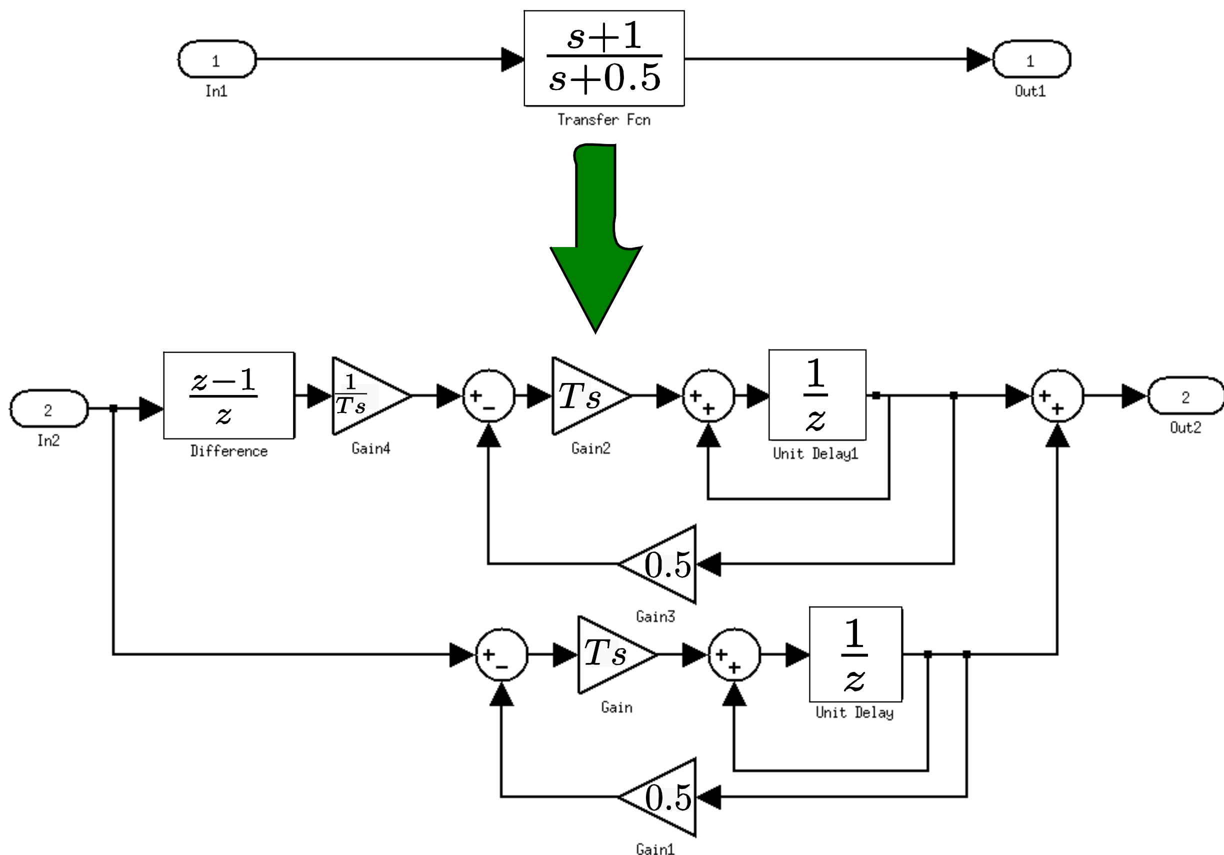

Figure 12. Continuous Transfer Function to Discrete-Time

VI. As we have demonstrated in the Scilab to C prototype, the translation of control proofs and properties requires the handling of the linear algebra types and operations such as vector, matrix, matrix product, and matrix transpose. Although both the *constant* block and the *product* block are in the subset of Simulink that S2L can translate, it is unclear whether or not the tool can translate a matrix constant or a product of matrices.

VII. The linear algebra types and operations are not supported natively in Lustre (unlike in Matlab), therefore we cannot perform a direct mapping from Simulink. Nor can we map the Lustre operators directly to the C functions implemented for the Scilab to C translator.

VIII. Blocks that are signal sources such as the step command input will also be translated since they are part of the stability property specification.

Lustre to C

Lustre to C represents the final step in the migration of control semantics from the design level down to the code level. The main problem now is not so much generating the C code from Lustre but translating the annotations expressing control semantics. SCADE already has a certified autocoder but we cannot adapt it for our purpose because it is a closed-source project. Unlike in our Scilab to C translator, the structure of Lustre does not allow a direct mapping to a library of C functions. This makes using the annotative template system described earlier difficult if not impossible. We have started looking at

tools such as Gene-Auto[18] which is a open-source code-generation framework for the translation of data-flow languages into imperative languages. At the moment Gene-Auto can handle 50 Simulink blocks and can output to C.

Conclusions

We have created an early prototype of Scilab to C autocoder in demonstration of generating C code with ellipsoidal invariants. This was done as part of the project to create a tool-set that can translate control system semantics from the design level to the source code level for the purpose of assisting formal verification. We hope that the creation of this *autocoder with proofs* will pave the path to a more rigorous guarantee of quality for safety-critical embedded control systems. From the prototype Scilab to C translator, we have further refined the language layers for the next prototype. An intermediate layer of Lustre will be added and the source language will be changed to Simulink with certain control annotation extensions. Future work includes completing the second prototype, testing the prototype on real control systems designs, adapting program analyzers and theorem provers tools to automate much of the verification of the annotated output.

References

[1] Aditya Agrawal and Gyula Simon and Gabor Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 - 56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[2] Gary J. Balas and John C. Doyle and Keith Glover and Andy Packard and Roy Smith. *μ-analysis and Synthesis Toolbox*. 1993.

[3] Patrick Baudin and Jean-Christophe Filliâtre and Claude Marché and Benjamin Monate and Yannick Moy and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. 2008. http://frama-c.cea.fr/acsl.html.

[4] S. Boyd and L. El Ghaoui and E. Feron and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15 of Studies in Applied Mathematics. SIAM, Philadelphia, PA, 1994.

[5] Caspi, Paul and Curic, Adrian and Maignan, Aude and Sofronis, Christos and Tripakis, Stavros and Niebert, Peter. From Simulink to Scade/Lustre to TTA: a layered approach for distributed embedded applications. *SIGPLAN Not.*, 38:153--162, 2003.

[6] Feron, E. From Control Systems to Control Software. *Control Systems, IEEE*, 30(6):50 -71, 2010.

[7] Heber Herencia-Zapana and Alwyn Goodloe. Formal Verification of Control Algorithms. Technical report, National Institute of Aerospace, 2011.

[8] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, 12:576--580, 1969.

[9] Ulf Jonsson and Chung-Yao Kao and Alexandre Megretski and Anders Rantzer. *A Guide to IQC-β: A MATLAB Toolbox for Robust Stability and Peformance Analysis*. 2004.

[10] Meenakshi, B. and Bhatnagar, Abhishek and Roy, Sudeepa. Tool for Translating Simulink Models into Input Language of a Model Checker. In Liu, Zhiming and He, Jifeng, editors, *Formal Methods and Software Engineering* in Lecture Notes in Computer Science, pages 606-620. Springer Berlin / Heidelberg, 2006.

[11] Yannick Moy and Claude Marché. *Jessie Plugin Tutorial, Beryllium version*. INRIA, 2009. http://www.frama-c.cea.fr/jessie.html.

[12] Owre, S. and Rushby, J. and Shankar, N. PVS: A Prototype Verification System. in Kapur, Deepak, Editors, *Automated Deduction* in Lecture Notes in Computer Science, pages 748-752. Springer Berlin / Heidelberg, 1992.

[13] Ana-Elena Rugina and Dave Thomas and Xavier Olive and Guillaume Veran. Gene-Auto: Automatic Software Generation for Real-time Embedded Systems. 2010.

[14] Timothy Wang and Romain Jobredeaux and E. Feron. A Graphical Environment to Express the Semantics of Control Systems. 2011. arXiv:1108.4048.

[15] V.A. Yakubovich. The S-Procedure in Nonlinear Control Theory. *Vestnik Leningrad University Math*, 4:73-93, 1971.

Acknowledgement

The authors would like to thank the United States Army Research Office for the MURI project. The authors would also like to thank the National Science Foundation, the Northeastern University, and the National Aeronautics and Space Administration for their financial support. The authors would finally like to thank Alwyn E. Goodloe of NASA Langley for all the good discussions on a variety of topics.

Appendix I

Linear Compensator in Scilab with Annotation of Control Semantics

```
1  Ac = [0.4990, -0.0500 ; 0.0100, 1.0000]
2  Cc = [564.48, 0]
3  Bc = [1;0];Dc = -1280
4  xc = zeros(2,1);
5  y=0
6  yd=0
7  receive(y,2)
8  receive(yd,3)
9  yc=3
10 P = [1e-3*0.6742, 1e-3*0.0428;1e-3*0.0428,1e-
       *2.4651];
11 while 1
12    //@ y^2 <=1
13    yc = max(min(y-yd,1),-1)
14    //@ y^2 <=1
15    u= Cc*xc + Dc*yc
16    //@ transpose(xc)*inverse(P)*xc<=1
17    xc = Ac*xc + Bc*yc
18    //@ transpose(xc)*inverse(P)*xc<=1
19    send(u,1)
20    receive(yd,2)
21  end;
```

Scilab to C Translator Output

```
1  #include "tools.h"
2
3  int main(int argc, char *argv[])
4  {
5    double** Ac_0;
6    double** Bc_0;
7    double** Cc_0;
8    double Dc_0;
9    double** P_0;
10   char false;
11   char true;
12   double u_0;
13   double** xc_0;
14   double** xc_0_temp;
15   double y_0;
16   double yc_0;
17   double yd_0;
18   Ac_0 = allocM ( 2, 2);
19   Bc_0 = allocM ( 2, 1);
20   Cc_0 = allocM ( 1, 2);
21   P_0 = allocM ( 2, 2);
22   xc_0 = allocM ( 2, 1);
23   xc_0_temp = allocM ( 2, 1);
24   Ac_0[0][0]=0.499;
25   Ac_0[1][0]=1.0e-2;
26   Ac_0[0][1]=-5.0e-2;
27   Ac_0[1][1]=1.0;
28   Cc_0[0][0]=564.48;
29   Cc_0[0][1]=0;
30   Bc_0[0][0]=1;
31   Bc_0[1][0]=0;
32   Dc_0=-1280;
33   xc_0 = zeros(2, 1);
34   y_0=0;
35   yd_0=0;
36   receive(y_0, 2) receive(yd_0, 3) yc_0=3;
37   P_0[0][0]=1e-3*0.6742;
38   P_0[1][0]=1e-3*4.28e-2;
39   P_0[0][1]=1e-3*4.28e-2;
40   P_0[1][1]=1e-3*2.4651;
41   while (1)
42
43   {
44
45 /*@ (y_0^2)<=1 */
46      yc_0=max(min(y_0+-yd_0, 1),-1);
47
48 /*@ (y_0^2)<=1 */
49      u_0=Cc_0[0][0]* xc_0[0][0]+Cc_0[0][1]*
            xc_0[1][0]+Dc_0* yc_0;
50
51 /*@ mult(mult(transpose(xc_0 ), 2, 1,inverse
     (P_0 )),1,2,xc_0,2,1)<=1 */
52      xc_0_temp = copy(xc_0,2,1);
53      xc_0 = add(mult
            (Ac_0,2,2,xc_0_temp,2,1,0,1) ,2 ,1,
            mult_scalar(yc_0, Bc_0,2,1,0),
            2,1,1,1);
54
55 /*@ mult(mult(transpose (xc_0 ),2,1, inverse
     (P_0 )), 1, 2, xc_0,2,1)<=1 */
56      send(u_0, 1)receive(yd_0, 2)   return 0;
57   }
58   return;
59
60 }
```