Automatic Fixed-point Code Generation for Modelica using Dymola

Ulf Nordström[1,2] José Díaz López[1] Hilding Elmqvist[1]
[1]Dynasim AB, Lund, Sweden, {Ulf.Nordstrom, Jose.Diaz, Elmqvist}@Dynasim.se
[2]Lund Institute of Technology, Lund, Sweden

Abstract

This paper describes a Modelica package for fixed-point arithmetics and automatic fixed point code generation for embedded systems and FPGA applications. Using Dymola [1] to investigate the dynamic behavior of the original model a fixed point representation is automatically generated. The model can then be simulated, using fixed point arithmetics to verify the fixed-point representation. Finally, code is generated for the desired target. Either integer C code for embedded systems or Mitrion-C code [2] for automatic VHDL code generation for FPGA targets.
Keywords: fixed-point arithmetics; automatic code generation; embedded system; DSP; FPGA; Mitrion-C

1 Introduction

Hardware-In-the-Loop simulations are widely used nowadays for design and testing of control systems in industry. Typical devices for such HILs are Digital Signal Processors (DSP) and Field Programmable Gate Arrays (FPGA). Typically, the development of algorithms or models is done in high level languages, not directly related to the target hardware. This is advantageous since the model keeps independent. But, specific code generation for the target platform has to be done.

Another important aspect is the following. During the development phase, floating point arithmetics is used for computations of algorithms and models. In many cases, the tolerances, characteristics of the system and performance of the target platform do not justify the use of such demanding floating point calculations. Furthermore, sometimes they are even an obstacle to HILs, since the computations are slower than required.

The possibility we explore in this paper is *fixed-point arithmetics* for modeling. We describe in this paper an aid platform for automatic code generation from Modelica models using fixed-point arithmetics.

In a first step, Dymola generates a corresponding Modelica model using fixed-point arithmetics. This model is intended for bit configuration testing and result assessment.

In a second step, the platform generates code in two variants

1. C code using exclusively integer data types for all variables with corresponding binary operation implementation. This type of code is intended mainly for DSP applications.

2. Mitrion-C code, using also integer data types, but with different word lengths. This code is mainly intended for FPGA applications.

As a scenario example, let us consider the task of developing an electrical control unit (ECU) for speed control of a simple vehicle drive train using DSP. To test different control strategies, a model of the closed loop system is implemented in Modelica. The model is depicted in Fig. 1.

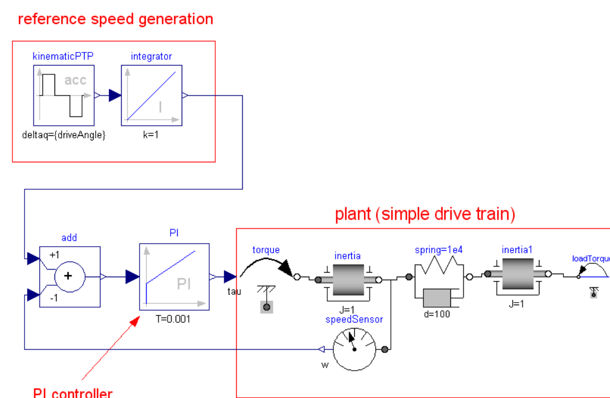

Figure 1: Simple drive train with PI speed controller in closed loop

After some experimentation we find appropriate parameter settings and verify the functionality of the controller. Our intention now is to realise the controller using an DSP.
Manually transforming the equations for the PI controller to a DSP program is a tedious and error prone task. In the following, we will see how the ModelicaFixedPoint package can be used to perform this task automatically.
Furthermore, the package provides error propagation analysis and fixed-point implementation of usual mathematical functions based on Newton-Rapson or table interpolation.

2 ModelicaFixedPoint package

The structure of ModelicaFixedPoint is presented in Fig. 2.

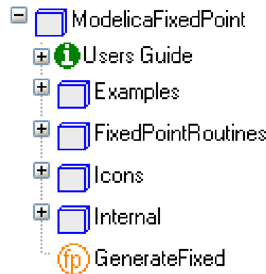

Figure 2: Library structure.

The main function of the library, `GenerateFixed`, is used for generation of the fixed-point representation and code generation. `GenerateFixed` preforms the 4 main steps of the conversion

1. Fixed-point translation
2. Range analysis
3. Precision analysis
4. Code generation

The internal symbolic engine of Dymola preforms symbolic manipulation of the original model and outputs a fixed-point converted model with all arithmetic operations replaced by function calls. Some examples of Modelica types and operations before and after conversion are shown in listings 1 and 2.

Listing 1: Modelica types and fixedpoint types

```
// before conversion
Integer x;
Real y;
Boolean z;

// after conversion
FixedPoint_Integer x;
FixedPoint_Real y;
FixedPoint_Boolean z;
```

The arithmetic operations are replaced by function calls and a unique identifier is inserted for every function call. *Type* can here be either Real or Integer depending on the types of the arguments, as example we present the basic operations.

Listing 2: Basic operations

```
// before conversion
a=u+v;
b=u-v;
c=u*v;
d=u/v;

// after conversion
a=Add_Type_Type(u, v, opId1);
b=Subtract_Type_Type(u, v, opId2);
c=Multiply_Type_Type(u, v, opId3);
d=Divide_Type_Type(u, v, opId4);
```

Consider now the PI controller of the ECU example. This controller is implemented in Modelica with the following equations

$$\dot{x} = \frac{u}{T}$$

$$y = k(x+u)$$

where x is the controller state variable, u is the input, k is the proportional gain and $1/T$ is the integral constant. Those equations converted to fixed point with ModelicaFixedPoint are presented in listing 3. The operator `FixedPoint2Derivative` is introduced to perform time integration in Dymola, during assesment of the fixed-point model.

Listing 3: Typical equations

```
der(x_aux)=FixedPoint2Derivative(Divide_Real_Real(u, T, op1));
y=Multiply_Real_Real(k, Add_Real_Real(x, u, op2), op3);
```

Range analysis
The proposed method uses a simulation based approach for determining ranges of variables and intermediate results in the equations. Simulating the original model and logging minimum and maximum values of all variables is important.This allows equation traversing and thereby compute accurate ranges for all intermediate results.
This method is more time consuming than methods like interval arithmetics [6] and affine arithmetics [4], but the resulting intervals are better.
ModelicaFixedPoint considers that every variable consists of *two parts*: an *integer* part and a *fractional* part. Both integer and fractional parts have a bit length

that sum up to the total *word length* used for the variable. The integer word length is denoted here by IW_L, the fractional word length by FW_L and the total word length by W_L.
The range information is used to assign the needed IW_L to all variables and operations.

Precision analysis
Precision analysis is done in two different ways depending on the target platform (DSP or FPGA). The two main parameters in `GenrerateFixed` are `wordLength` (target platform word length) and `RTOL` (relative tolerance required for conversion). The function dialog box is shown in Fig. 3.

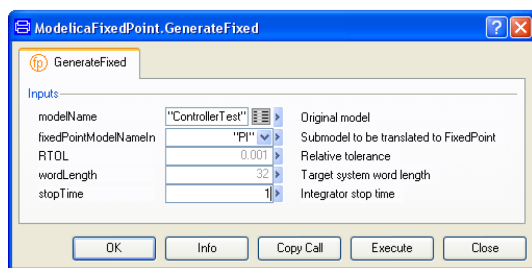

Figure 3: Dialog box.

Specifying `wordLength` indicates that integer C code is to be generated for a DSP target. The specified word length then sets a unique word length for the entire system and the FW_L is set to maximize precision given by `RTOL`.
For FPGA targets the task becomes more complicated due to the ability to use different W_L in the system. Our approach is to use a backward error propagation scheme based on the error propagation analysis in section 5 to determine all FW_L and shift operations from a user specified tolerance `RTOL` at the output of the system.

The package `FixedPointRoutines` is also included. This package contains fixed-point arithmetic functions and records for simulation of fixed-point Modelica code. The structure is shown Fig. 4.

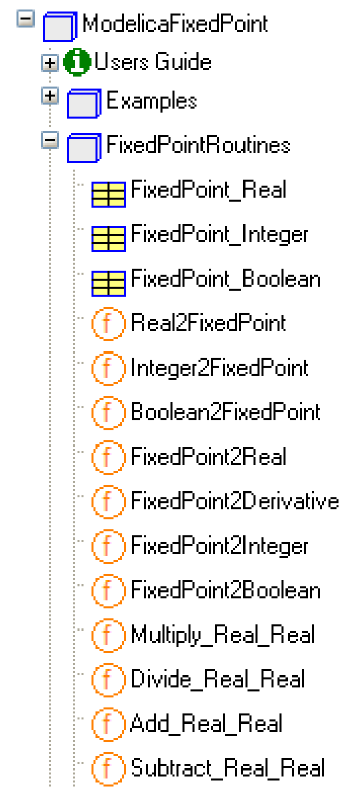

Figure 4: FixedPointRoutines.

3 Fixed-point scenario

Integer arithmetic operations execute much faster than their corresponding floating-point operations because of their simplicity. In the case of FPGA, silicon surface area and power consumption are also significantly reduced using integer arithmetics. On the other hand, DSP devices come normally with simple arithmetic logic units, missing completely hardware implementation of floating point arithmetics.

The achievable precision using integer arithmetics is closely related to the architectural word length of the target platform. In the case of DSP, the integer data word size, typically 16, 24 or 32 bits, limits the possible precision. For FPGA, there is the possibility of adapted word length for each operation. However, this migth exceed practical limitations. A tradeoff must be considered, minimizing the number of bits used while not violating constraints on precision. Another interesting aspect of the FPGA is parallelism. The programmable structure of the FPGA allows exploiting independent computations, by implementing those in parallel structures.

The target language for FPGA is Mitrion-C. This language is a dataflow language with a syntax resembling C, developed by Mitrionics AB in Lund. The generated Mitrion-C code is compiled into a configuration of the Mitrion Virtual Processor. The Mitrion Virtual Processor is a fine grain massively parallel softcore processor which can be downloaded and ran on a variety of FPGA's, see [2] for details.

4 Fixed-point Arithmetics

From a hardware point of view, fixed-point arithmetics is essentially integer arithmetics with bit shifting. Using integers to represent non-integer values is done by considering an imaginary binary point as follows. Consider the binary representation of an integer in Natural Binary Code (NBC).

$$(b_n, b_{n-1}, \ldots, b_2, b_1, b_0) = \sum_{j=0}^{n} b_j \cdot 2^j,\ \forall b \in \{0,1\}. \quad (1)$$

Now, using the same set of bits to represent a non-integer value can be done by placing a binary point between $i-1$ and i. Thus

$$(b_n, \ldots, b_{i+1}, b_i, b_{i-1}, \ldots, b_0) = \sum_{j=0}^{n} b_j \cdot 2^{j-i}. \quad (2)$$

The fixed-point representation in (2) with the binary point between $i-1$ and i is said to have i bits of precision. By q we denote the value in NBC, i.e. the value seen by the arithmetic unit. The smallest representable number is the so-called the resolution of the representation and is equal to 2^{-i}. We define integer word length, $IW_L = (n+1) - i$, and fractional word length, $FW_L = i$. The word length, that is, the total number of bits used, is denoted by W_L and is equal to $n+1$.

Converting a floating-point number y to fixed-point representation q is done by

$$q = \lfloor 2^i \cdot y \rfloor, \qquad i, q \in \mathbb{Z}, \quad y \in \mathbb{R}, \quad (3)$$

where $\lfloor \cdot \rfloor$ denotes rounding towards floor.

The basic operations on fixed-point numbers; addition, subtraction, multiplication and division, are implemented using ordinary integer operations and bit shifting. The bit shifts (left shift and right shift) of a fixed-point number q are

$$(q << i) = q \cdot 2^i \quad (4)$$

and

$$(q >> i) = q \cdot 2^{-i}. \quad (5)$$

Bit shifting is used extensively to align binary points and to rescale variables. The shift operators are used to rescale both the inputs of an operation and the output. Consider a binary operation op on two fixed-point variables q_1 and q_2, called operands. The implementation of such operation is

$$\mathrm{Op}(q_1, q_2, s) = \big((q_1 << s_1)\mathrm{op}(q_2 << s_2)\big) << s_3 \quad (6)$$

where op $\in \{+, -, *, /\}$, or equivalently

$$\mathrm{Op}(q_1, q_2, s) = \big((q_1 \cdot 2^{s_1})\mathrm{op}(q_2 \cdot 2^{s_2})\big) \cdot 2^{s_3}. \quad (7)$$

Clearly, associating a 3-tuple of shifts $s = \{s_1, s_2, s_3\}$ with an operation we define the syntax of the basic operations. For readability reasons we will use the symbol for left shift, $<<$, followed by s_i to denote shifts. When $s_i > 0$, the shift is left and when $s_i < 0$ the shift is right.

For addition and subtraction the binary point must be aligned before the operation. There will be a loss of precision in the result when right shifting is used. Thus, it is preferable to use left shifts when possible. However, right shifts may sometimes be needed in order to avoid *overflow*. Adding or subtracting numbers with very large difference in magnitude does not mean any problem. The smaller one will naturally be numerically insignificant compared to the larger one.

Multiplication and division are more difficult to implement. Multiplying two fixed-point variables each having W_L number of bits will generate a result having $2W_L$ number of bits. This is likely to cause an overflow.

Using the shift operators on the variables prior to the fixed-point multiplication, one could shift the operands so that no overflow can occur. For the situation above, the secure shiftings correspond to right shifting each of the operands by $W_L/2$ bits prior to the operation. This reduces of course the final resolution of the result.

Division has a similar problem. Dividing two variables with FW_L fractional bits generates a result having no fractional bits. Here we loose resolution unnecessarily. One solution is to right shift the denominator prior to the division, keeping as much precision as possible.

For operations other than the basic binary ones, there are often no straight forward implementation. Conditional operations such as $\{<, \leq, >, \geq, ==, \neq\}$ are an exception. These operations on fixed-point numbers are the same as their floating-point counterparts although the binary points of the operands must be aligned before evaluation. This alinement of binary points can reduce resolution and therefore cause the wrong conditional branch to be evaluated.

For other functions we use linear interpolation in tables or Newton-Raphson iterations to evaluate the result, as mentioned earlier. With information on the range of the variable and the result, tables covering the entire range with appropriate resolution are generated. Development of interpolation tables and Newton-Raphson iterations is currently ongoing.

In order to use fixed-point arithmetics efficiently one need to find appropriate shifts, and W_L in case of FPGA, for all variables and operations. The automatic *floating-point to fixed-point conversion problem* can then be summarized as finding IW_L and FW_L for all variables and respective appropriate shifts for all operations such that no overflow occurs, quantization errors are kept low and the total W_L is kept as low as possible.

5 Error propagation

Error propagation analysis has to be done to solve the fixed-point conversion problem, and assign FW_L to variables and operators. For further theory details, see [3].

A throughout error analysis would require *a priori* knowledge of all shifts and FW_L. For simplicity, we assume that all intermediate results can be stored, eliminating the the risk of overflow. Clearly and in practice, this assumption is relevant and valid for Mitrion-C code only.

For addition and subtraction the function implementation is eq. (7). Multiplication and division are somewhat different.

For multiplication, the possibility of storing the intermediate result eliminates the shifting of the operands before the operation. Instead, the result has to be shifted to the appropriate FW_L.

For division, the assumption avoids shifting the denominator. Hence, only the numerator and the result are shifted to the appropriate FW_L at the output.

Conversion

The conversion to fixed-point causes an error, the so called *conversion error* or *quantization error*, denoted by δ. We have

$$|\delta| = |y - \tilde{y}|, \qquad \delta \in \mathbb{R}. \tag{8}$$

where y is the floating-point value before conversion and $\tilde{y}$ is the recovered floating-point value after conversion. From [3], the conversion error is bounded by

$$|\delta| < 2^{-i}. \tag{9}$$

Defining

$$\Delta = \sup|\delta| = \sup|y - \tilde{y}|, \tag{10}$$

the results of the error propagation analysis will be presented.

Addition and subtraction

Addition and subtraction have the same error propagation properties. Considering the absolute error for addition we have

$$\Delta_R = \Delta_1 + \Delta_2 = 2^{-i_1} + 2^{-o} + 2^{-i_2} + 2^{-o}. \tag{11}$$

where i is the resolution of the operand and $o = FW_L$ is the resolution of the result.

Multiplication

Considering the relative error for multiplication, we have

$$\left|\frac{\Delta_R}{R}\right| = \sup\left|\frac{\delta_R}{R}\right| = \left|\frac{2^{-i_1}}{y_1}\right| + \left|\frac{2^{-i_2}}{y_2}\right| + \left|\frac{2^{-o}}{y_1 y_2}\right| \tag{12}$$

where y_i is the largest value taken by each variable respectively.

Division

Again considering the relative error we have

$$\left|\frac{\Delta_R}{R}\right| = \left|\frac{2^{-i_1}}{y_1}\right| + \left|\frac{2^{-i_2}}{y_2}\right| + \left|\frac{2^{-i_2+o}}{y_1}\right|. \tag{13}$$

The error propagation is different from the multiplication.

6 Code generation

The information gained in the fixed-point conversion and analysis is used to generate code for different target platforms. In this paper we mainly focus on generating integer C code typically used in DSP's. But, also Mitrion-C code for FPGA's is possible with appropiate error propagation analysis.

For easy interaction with standard generic integration routines, all variables are categorized and mapped to a set of vectors, as in table 1. All right-hand-side equations are gathered in one function, called *rhs-function*. This encapsulation allows code portability.

This encapsulation and code generation is used in cases where only a subsystem is intended for fixed-point representation. Then we substitute that particular part of the system with a function call and the rest of the system is treated as usual.

To avoid unnecessary computations, constants are shifted according to the fixed-point representation and evaluated during the code generation.

Generation of integer C code

Currently, the generated C code supports a format for easy interaction with Dymola to make it easy to validate the functionality of the generated code. Different

x	state variables
x_dot	derivatives of state variables
w	auxiliary variables
u	input variables
y	output variables
p	parameters

Table 1: Table of categorized variables

DSP targets could require different structures, mainly of the function call, i.e. the body of the function would remain the same.
The syntax of the Modelica interface to the rhs function is in listing 4.

Listing 4: Modelica Interface

```
// Modelica interface
function Name
  input Integer n;
  input Integer t;
  input Integer x[:];
  input Integer x_dot[:];
  input Integer w[:];
  input Integer u[:];
  input Integer y[:];
  input Integer p[:];
  input Integer idemand;
  output out_w[size(w,1)];
  output out_x_dot[size(x_dot,1)];
  output out_y[size(y,1)];
  external "C";
  annotation(Include="#include<Name.c>");
end Name;
```

and the function itself in listing 5.

Listing 5: C interface

```
// C interface
int Name(int n, int t, int *x, int size_x,
         int *x_dot, int size_x_dot, int *w,
         int size_w, int *u, int size_u,
         int *y, int size_y, int *p, int size_p,
         int idemand, int *out_w, int size_out_w,
         int *out_x_dot, int size_out_x_dot,
         int *out_y, int size_out_y){

   /* BODY OF FUNCTION */
}
```

The main body of the function is the section containing the equations for computing the auxiliaries, derivatives and outputs of the system. The basic operations, addition, subtraction, multiplication and division, uses basic integer operations and the appropriate shifts are inlined in the code according to the syntax in (6). The integer C code of the equations in listing 3 can be seen below in listing 6.

Listing 6: Integer operations in C

```
/* Compute derivatives */
x_dot[0]=((u[0])/(p[1]>>15)<<10);

/* Compute outputs */
y[0]=((p[0]>>16)*(((x[0])+(u[0]>>1))>>15)<<1);
```

Conditional operators and the *IfThenElse*-statement are also inlined with appropriate shifts. Essential for the conditional operators are that the binary points are aligned before evaluation. Hence, also here, shift operations are inlined in the code. An example can be seen in listing 7.

Listing 7: Conditional operations in C

```
// Computing "z = if x<y then x-y else y-x;"

w[0] = ((((x[0]>>6)<(x[1]))==1)?(((x[0]>>6)-
       (x[1]))):(((x[1])-(x[0]>>6))));
```

A C library for special fixed-point functions is under development for functions such as *min*, *max* and *abs*. Also for these functions, except for abs, the binary point must be aligned prior to the evaluation. This can be solved by having the appropriate shifts inlined in the function call. For other functions, such as *trigonometric* functions and *square root*, code for linear interpolation tables or Newton-Raphson iterations schemes will be generated. This is still in the development phase although successful implementation of linear interpolation using only integer arithmetics have been implemented using Modelica.

Generation of Mitrion-C code

Although the syntax of Mitrion-C resembles that of C, the generated code is quite different. This is mainly due to the possibility of having a mixture of different word lengths in the system. This allows us to allocate resources (word length of the data path) were they are needed the most.
The result of this is that it is no longer possible to inline any operations or shifts. Instead a unique function is generated for every operation. All shifts and word length declarations are hard coded in these functions.

7 Examples

7.1 Speed control with PI controller

Using the library presented in this paper lets us automatically generate a fixed-point model of a PI controller using various word lengths. Simulating the system using the fixed-point PI controller we can verify the functionality of the controller and also get hints regarding the word length needed to fulfill the specifications.
After some simulations, depicted in Fig. 5, we conclude that a word length of 8 bits is enough for our application. This results in a smaller/cheaper DSP for the ECU instead of the one originally intended. Also, integer C code for the controller is automatically generated.
For Modelica users this is a familiar way of working, implementing systems and algorithms at high abstraction levels by drag and drop of predefined components

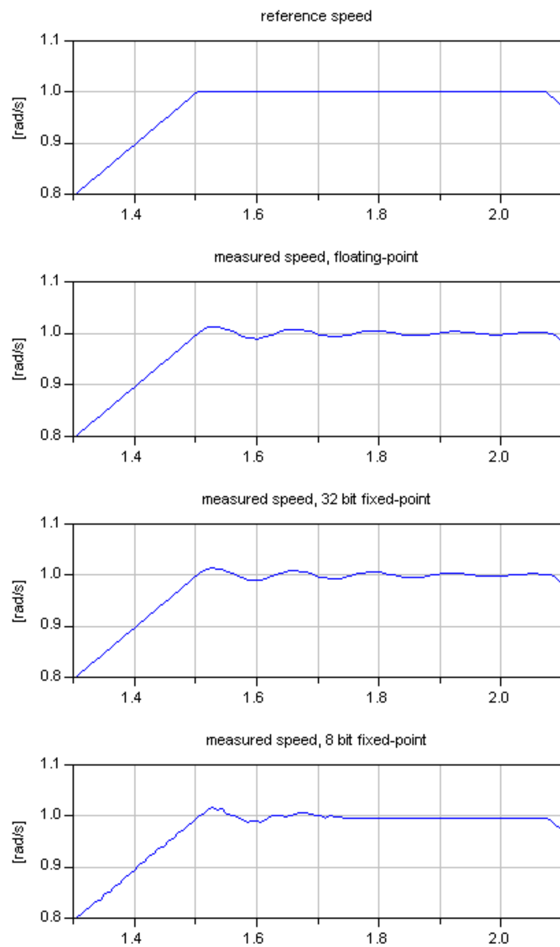

Figure 5: Plot of reference speed and measured speed using PI controllers.

in a diagram or manually typing Modelica code. With this library we aim to keep this way of working while extending it by automatically generating fixed-point code for external computational devices. The following example is the opposite situation: plant realisation with fixed point arithmetics.

7.2 ServoMotor with FixedPoint

This example explores the plant realisation using fixed point arithmetics. The situation is a simple electric drive used to place a inertia at a given angle. The simple model of electric drive is depicted in Fig. 6. The model consists of an ideal EMF device with shaft inertia and electric inductance and resistance.

We provide the model with the load and a gearbox for better precision on the load. The system is encapsulated in a model with inputs and output as depicted in Fig. 7.

Finally, we choose a PID regulator to control the angle

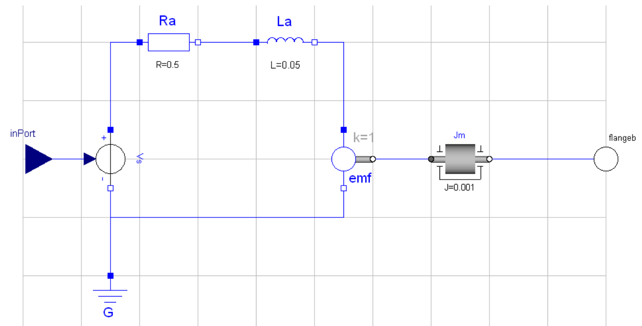

Figure 6: Electric Drive Model

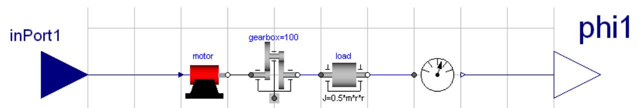

Figure 7: Electric Drive and load

of the load and set it to the reference. The final model is depicted in Fig. 8.

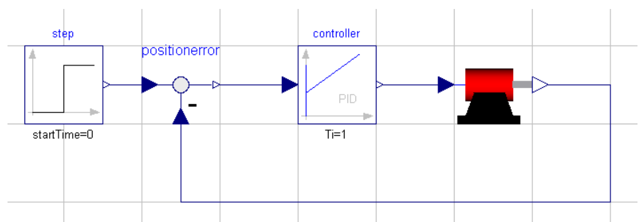

Figure 8: PID regulator to position the inertia to build the servo drive

The question now is if there is a cheap and fast fixed-point representation of the system, good enough to use with the PID regulator in hardware. After Fixed-point trasnalation, we get with the tool the system in Fig. 9.

The first attempt is done analysing `servoMotorWithLoad` model within the time interval $[0,1]$ and for 8, 16 and 24 bits. The result of the simulations in Dymola are depicted in Fig. 10. We observe that even though we have more precision, the steady state is never really reached. The solution becomes oscillatory.

The first though may be that the number of bits of the fixed point representation is not high enough to make the system reach steady state. This is not the case. What is happening is that the precision analysis is tightly adjusting the fixed-point representation to the time interval and span interval of every variable.

It is possible to resemble the correct dynamics of the system with just 8 bits. The result is depicted in Fig. 11. The only difference is that we performed the precision analysis in the time interval $[0,100]$ instead.

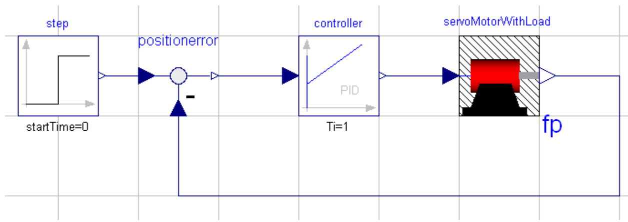

Figure 9: Fixed Point version of the motor with load.

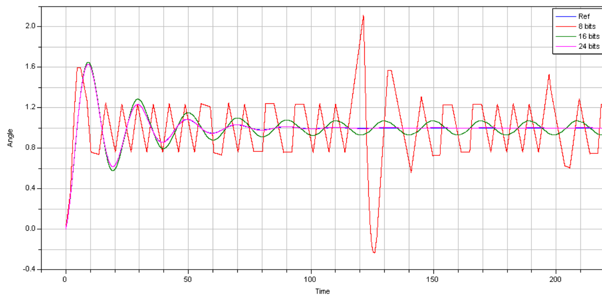

Figure 10: Simulation result with model analysed in time interval $[0,1]$

We can see the C code generated for the example also in listing 8. Notice the implementation of the fixed-point operations using shifts. This implementation can be also encapsulated to generate code to link with external fixed-point libraries.

Listing 8: C function generated

```
/* BODY OF THE FUNCTION */
/* Compute Auxiliary variables */
    if (idemand == 3) {
        w[0] = 0;
        w[1] = 0;
        w[2] = 0;
        w[3] = 0;
        w[4] = 0;
        w[5] = 0;
        w[6] = 0;
        w[7] = ((32>>4)*(((((m>>4)*(r>>3))>>4)*(r>>3))>>4)<<2);
        w[8] = ((motor_emf_k>>3)*(x[1]>>4)<<1);
        w[9] = ((gearbox_ratio>>3)*(derload_w_aux>>4)<<2);
        w[10] = ((w[8])-(((motor_Jm_J>>3)*(w[9]>>4)))<<1);
        w[11] = (-(((gearbox_ratio>>4)*(w[10]>>3))));
        w[12] = ((motor_Ra_R>>3)*(x[1]>>4));
        w[13] = ((u[0])-(w[12]>>5));
        w[14] = ((gearbox_ratio>>4)*(x[0]>>3)<<1);
        w[15] = ((motor_emf_k>>3)*(w[14]>>4)<<1);
        w[16] = ((w[13])-(w[15]>>2));
        w[17] = ((gearbox_ratio>>4)*(x[2]>>3)<<1);
        w[18] = (-(((w[10]>>7)+(w[11]))));
        w[19] = 0;
    }

    /* Compute derivatives */
    if (idemand == 2) {
        x_dot[0] = ((((gearbox_ratio>>4)*(w[8]>>3)))/(((w[7])+
                   (((((motor_Jm_J>>3)*(gearbox_ratio>>4)>>3)>>3)
                   *(gearbox_ratio>>4))<<4))>>1)<<7);
        x_dot[1] = ((w[16])/(motor_La_L>>3)<<3);
        x_dot[2] = x[0];
    }

    /* Compute outputs */
    if (idemand == 1) {
        y[0] = x[2];
    }
```

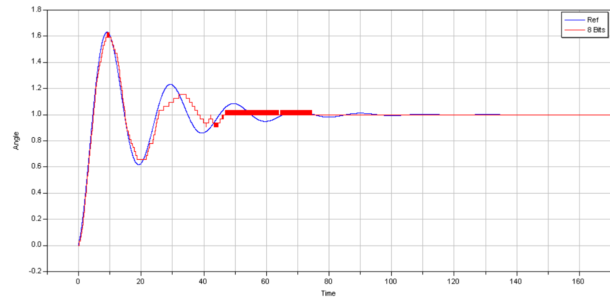

Figure 11: Simulation result with model analysed in time interval $[0,100]$

8 Summary

We have presented a tool that enables a developer to use Modelica models for code generation for an embedded system or FPGA with a minimum of manual interaction.

References

[1] Dymola, Dynasim AB, www.dynasim.com

[2] Mitrion-C, Mitrionics AB, www.mitrion.com

[3] Ulf Nordström. **To be published**. Automatic Fixed Point Code Generation in Modelica using Dymola. Lund, Sweden: Master's thesis, Department of Automatic control, Lund Institute of Technology, 2006.

[4] Claire F.Fang, Rob A Rutenbar, Tsuhan Chen. Fast, Accurate Static Analysis for fixed-point Finite-Precision Effects in DSP Designs. Pittsburgh, USA, Department of Electrical and Computer Engineering, Carnegie Mellon University.

[5] Float-to-Fixed Conversion Tool, www.float-to-fixed.com

[6] R. E. Moore. Interval Analysis. Prentice-Hall, 1966.

[7] The Mathworks, Simulink fixed-point, 2005.