A Model-in-the-Loop Technique for Digital Twin Simulations (MIL-DT): A Generalized Approach for Simulating Embedded Software

B Morris[a*],T Cheng[a], C Koontz[b]
[a]*L3Harris;* [b]*Bechtel*
*Corresponding author. Email: Benjamin.Morris@L3Harris.com

Synopsis

This article presents a Model-in-the-Loop approach to Digital Twin simulations (MIL-DT) for cases where auto code generation or high level software modelling is not possible due to complexity or toolchain restrictions. A high fidelity "twin" of the hardware / software interaction is instead created by parsing a portion of the embedded source code into a MATLAB / Simulink graphical diagram. The result is then simulated alongside a hand coded model of the underlying hardware, also in MATLAB / Simulink.

This modelling approach is an alternative to auto code generation, which also enables an accurate model of hardware / software interaction. In auto code generation a hand coded feedback control diagram is parsed to create embedded software that is used with minimal modification in a unit under test. This is typically done with the assistance of a commercial toolchain such as MATLAB.

The MIL-DT model proposed here model consists of three main parts: a software application layer, a software support layer, and a physics based model of plant hardware. To create such a model the target codebase is divided into two parts, a software application layer and a support layer, which are separated by an application programming interface (API). The application layer is imported or compiled in an automated process and is a highly accurate model of the software itself. The support layer is written to emulate (or "stub") any function calls or operations needed by the application layer. The API enforces a structured interaction of data communication and function calls. Finally, the physics based hardware model is coded to contain as much detail as necessary to enable risk reduction in the end application.

The MIL-DT approach is illustrated in a case study of embedded feedback control of a DC-DC power converter. In this case study a simple C file is parsed into a Simulink diagram, which makes up the application layer of the MIL-DT. The support layer consists of the Simulink blocks themselves, and the physics layer is a full switching model of the DC-DC converter. The case study shows how an MIL-DT model can be used to tune the expected response of a system by modifying the original source, importing the application layer, and re-running the resulting model.

Keywords: Digital Twin; Model in the Loop; Simulation; Software Automation; Power Conversion

1 A Review of Digital Twin Simulations in the Context of Existing Modelling Approaches

The usage of digital twin simulations for engineering applications (Kumar et al., 2021) broadly mirrors existing Continuous Integration (Shahin et al., 2017) and automated unit testing techniques (Williams et al., 2009) used in pure software development. At the time of writing there are several interpretations of what the term "digital twin" may mean, from multiphysics simulations (Magargle et al., 2017), to a wide range of concepts in manufacturing (Kritzinger et al., 2018), to certain types of model based systems engineering (Madni et al., 2021).

One of the earliest usages of the term "digital twin" is in a NASA technology roadmap (Shafto et al., 2010), quoted below, which refers to a specific type of high fidelity probabilistic model that is continuously updated with sensor data:

> A digital twin is an integrated multi-physics, multi-scale, probabilistic simulation of a vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its flying twin ...
>
> In addition to the backbone of high-fidelity physical models, the digital twin integrates sensor data from the vehicle's on-board integrated vehicle health management (IVHM) system, maintenance history, and all available historical/fleet data obtained using data mining and text mining.

Authors' Biographies

Benjamin Morris is a Lead Engineer at L3Harris in Anaheim, CA. His current work is in software development for the simulation, optimization, and feedback control of electrical power conversion systems. He has received an MS and Ph.D. in Electrical Engineering from the University of Michigan, Ann Arbor, and a BS in Mechanical Engineering from the University of Illinois at Urbana-Champaign.

Timothy T. Cheng is the Software Engineering Lead for MPES Division, L3Harris. He is responsible for all software engineering and algorithmic modeling for products involving Maritime Power and Energy Solutions. He brings over thirty years (with twelve management years) of software engineering experience on diverse products for major corporations as well as venture backed startups. He has received a BS in Aerospace Engineering from the University of Michigan, Ann Arbor and an MBA from Arizona State University in Tempe.

Cody Koontz is a senior engineering manager for a division of Bechtel in Pittsburgh, PA. The group he currently leads is responsible for developing and fielding power electronic equipment, as well as technological advancements in the field. He has received an MS in Electrical Engineering from the University of Pittsburgh and a BS in Electrical Engineering from Grove City College.

It is important to note that although a wide variety of literature has appeared around different concepts mentioning "digital twin", that no single formal definition has yet emerged. A recent draft report by the National Institute of Standards and Technology (NIST) reaches this same conclusion (Voas et al., 2021):

> Currently, there are several unofficial "definitions" for digital twins—some created by researchers, some by standards committees and consortia, some by industry, and still others that are implicitly suggested by commercial enterprises that make statements that their software applications are "digital twin-compliant" in spite of the absence of any agreed-upon definition or consensus of vision for digital twins. Despite today's nebulous understanding and lack of formal definition of what digital twins really are—or will eventually become—the definition of a digital twin used in this paper is:
>
> A digital twin is the electronic representation—the digital representation—of a real-world entity, concept, or notion, either physical or perceived.

In the spirit of the NIST definition, this paper introduces a specific type of digital twin simulation: the Model-in-the-Loop Digital Twin (MIL-DT) which consists of three main parts: a software application layer, a software support layer, and a physics based model of plant hardware. This is achieved by an automated import process that creates a copy of the software application layer that is suitable to be compiled and or run in the modelled environment. The auto generated code is not otherwise modified by hand after the import; to do so would mean that the imported application is no longer a twin of the target system. Thus an MIL-DT seeks to create a twin of the application layer software in a target system.

The primary benefit of an MIL-DT is that it provides an automated process to evaluate software in simulation before conducting physical testing, thus reducing the demand on limited hardware availability. An additional benefit of an MIL-DT is the opportunity to apply automated unit testing and Continuous Integration (CI) concepts to evaluate the closed loop behavior of the software. One limitation of an MIL-DT is that the software development team must manually write stub functions to support all API calls of the application layer. This results in greater nonrecurring engineering costs (NRE) when a digital twin model is created, and additional costs to maintain the API when the underlying hardware or firmware changes.

1.1 Comparison with Model-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop Techniques

In order to put the MIL-DT approach into context it will be briefly compared against more traditional modelling and testing paradigms such as Model-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop. Although the usage of these terms sometimes varies, this high level classification can still provide helpful insights. For additional information and distinctions betwen model types, see (Nibert et al., 2012).

Model-in-the-Loop simulations are physics based or behavioural simulations where all aspects of the plant and feedback controller are virtual, including plant hardware, external noise sources, sensors, effects of sampling, etc. These types of simulations are typically run on a single processor in a commercial simulation environment such as MATLAB / Simulink, Labview, Amesim, or others. There is not a need at this point to choose a coding language for the end application, or a processor family, or to address processor related issues such as caching, frame rates, or synchronization, since all aspects of the simulation are virtual. Model-in-the-Loop simulations may be run faster than real time or slower than real time, depending on the details of model itself.

Software-in-the-Loop simulations contain actual software intended for the target environment, typically incorporated in one of two ways: the first is where embedded software is compiled and run within an otherwise pure simulated environment; and the second is where software is auto generated from the model using a commercial or custom utility. These simulations may be run faster than real time or slower than real time, similar to Model-in-the-Loop simulations.

Hardware-in-the-Loop simulation / testing involves some aspect of actual physical hardware, sometimes including sensors, processors, fault injection mechanisms, or other similar methods to incorporate the physical world into the simulation. These types of simulations / tests are most effective when the virtual components are run in a hard-real-time environment, otherwise the realism introduced by physical hardware would be distorted.

1.2 Additional Commercial Software Approaches to Model Development

Note that the Mathworks offers commercial automatic code generation tools (Krizan et al., 2014) and automatic software verification tools (Börcsök et al., 2009) that solve a similar set of problems as the MIL-DT approach proposed here. Companies such as DSpace and OpalRT, and realtime Linux distributions such as ROS (Robot Operating System) also offer software solutions to address issues similar to the ones presented here.

In many cases however a non-commerical option may be preferred. The code that is created by auto code generation toolchains may not conform to a desired coding standard, and the need to modify this code after the

auto generation stage may limit the developer's ability to simulate the end product effectively. Since there may be an underlying need for the codebase to have a lifetime that exceeds the support a single commercial product, a non-open-source solution may be preferred. These types of solutions are often used in defence applications to avoid potential liability introduced from open source projects.

One main benefit of the digital twin approach proposed here is that the original toolchain for authoring and compiling code is left unmodified. Note that in Figure 1, the digital twin models are created from an already existing codebase. Instead of relying on commercial tools for exporting software from models, custom tools are written that create runnable Simulink models from an existing source. The Mathworks offers a wide variety of command line and scriptable tools that can be used to create Simulink files from code, markup language, or other sources of data.

2 Creating a Model-in-the-Loop Digital Twin (MIL-DT) Model

As mentioned earlier, the MIL-DT consists of three main parts: a software application layer (that is imported from existing code), a software support layer (that is hand coded), and a physics based model of plant hardware (that is also hand coded). The detailed process to create the model is as follows, and will later be illustrated in an extended case study:

1. Separate the codebase into an Application Layer and a Support Layer.
    - Application layer software consists of those functions that will have a "twin" in the model. These aspects of software will be imported with the highest level of detail, mimicking both functionality and implementation details. This typically includes filter structure and coefficients, state machine definitions, and signals in engineering units.
    - Support layer software is everything else that the application layer needs to function properly, which may not necessarily be directly relevant to the behaviour of interest. These typically include mathematical support libraries, details of filter implementations, or file input-output utilities.
2. Identify hardware components of the target system that are relevant to the behaviour of interest which will be modelled in the MIL-DT simulation as a physics based plant model.
3. Automate parsing or importing of the application layer software.
4. Code or implement an API so that the application layer software can utilize necessary functions in the software support layer. The API provides data routing, synchronization, and a structured way for the application layer to call functions in the support layer.
5. Code and run simulations that automate the models to determine if relevant requirements have been satisfied.

The relationships between the original embedded source code, the application layer model, and the support layer software model are shown in Figure 1. The box in the upper left represents all aspects of the embedded source, including hand written code, script generated code, configuration data stored in a database or markup files, parseable diagrams, and any other artifact that could loosely be considered "source code".

In most cases, engineering judgement is needed to decide which aspects of source code should be automatically imported into the application layer, and which should be modelled by hand in the support layer. The level of engineering effort associated with one choice or another can be significant.

3 Case Study: An MIL-DT model of a Simplified DC-DC Power Converter

The following subsections illustrate the MIL-DT modelling process on a simplified buck converter and will mirror the methodology presented in Section 2. This case study was designed such that both the hardware design and the feedback regulator are highly simplified, which serves to focus attention on the methods used to create various aspects of the model. See Figure 2 for a schematic diagram of a basic synchronous buck converter. The design parameters are given in Table 1.

Depending on the power range and level of criticality of the system, a wide variety of software subsystems are needed to support a system such as the one above, despite the fact that the schematic itself is highly simplified. The overall codebase will often include functions for a user interface, logging, safety overrides, state machines enabling orderly startup and shutdown, support for gate drivers and modulators, and real-time controllers that actually carry out the regulation. The relevant source code is provided in "main.c" and "regulator.c". See Table 2 and Table 3. In this example only these two files are provided. Although these functions are straightforward they contain or reference a number of functions that are not directly provided. Nevertheless, purpose of these functions can be inferred even if the exact implementation is not provided. See Table 4 for a full list of dependencies. The

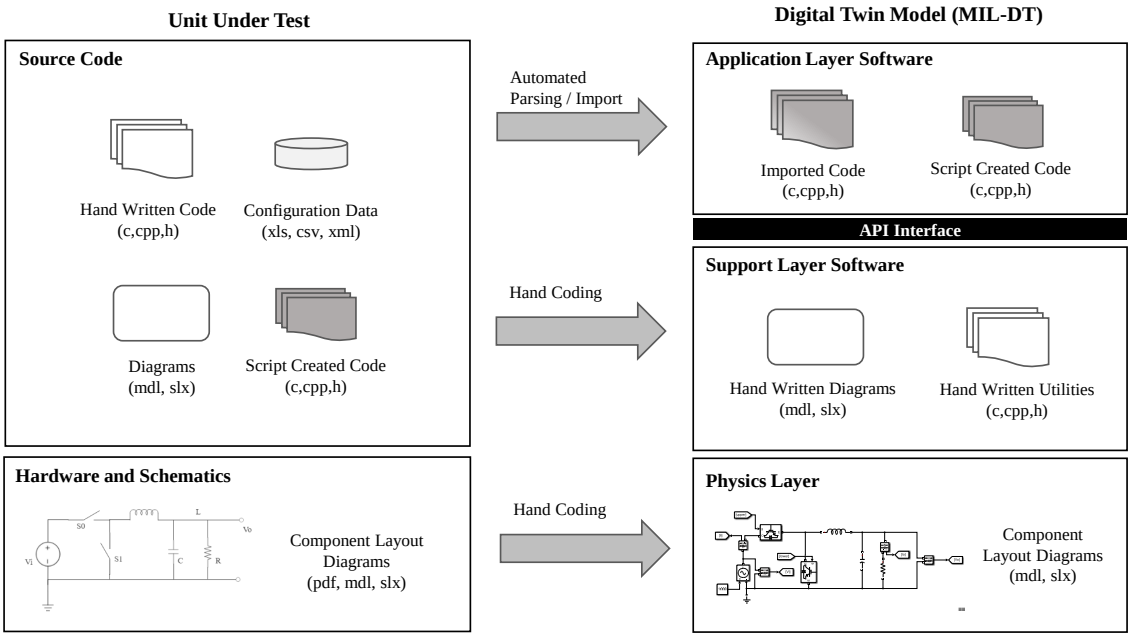

Figure 1: Relationships between the Software and Hardware of the Unit Under Test and the corresponding Digital Twin Model (MIL-DT).

Table 1: Design Parameters for the Synchronous Buck Converter

Design Parameter	**Value**
Input Voltage V_i	1000 V
Output Voltage Setpoint V_o	200 V - 400 V
Switching Frequency	5000 Hz
Inductance L	10^{-3} H
Capacitance C	10^{-5} F
Resistance R	1Ω

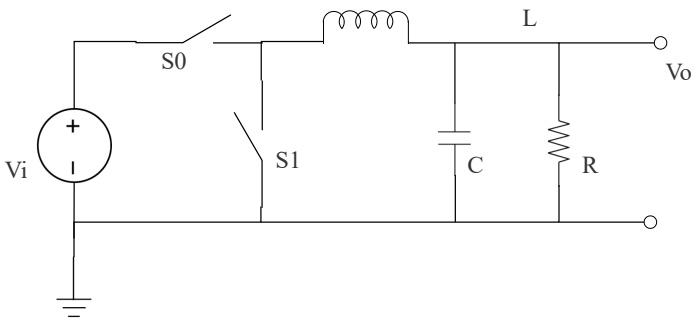

Figure 2: Schematic of a Simplified Synchronous Buck Converter

Table 2: Embedded Source: Original Feedback Regulator (above) and Updated Feedback Regulator (below)

regulator.c (original)

```
double regulator(double Vsetpoint, double Vin, double Iin, double Vout,
      double Iout, double ts) {

    double err, term1, term2; // intermediate variables
    double maxModIndex = 0.45, minModIndex = 1e-3;
    double Kp = 1e-3, Ki = 1e-3, Kd = 1e-6;
    static integratorStateVars intStates;
    static derivativeStateVars derivStates;

    // perform the calculations
    err = Vsetpoint - Vout;
    term1 = Kp*err + Ki*integrate(err, intStates) +
      Kd*derivative(err, derivStates);

    modIndex = max(min(term1,maxModIndex),minModIndex);
    return modIndex;
}
```

regulator.c (updated)

```
double regulator(double Vsetpoint, double Vin, double Iin, double Vout,
        double Iout, double ts) {
    double err, term1, term2; // intermediate variables
    double maxModIndex = 0.45, minModIndex = 1e-3, minVin = 500;
    double Kp = 1e-3, Ki = 1e-2, Kd = 1e-6, Kfeedforward = 0.5;
    static integratorStateVars intStates;
    static derivativeStateVars derivStates;

    // perform the calculations
    err = Vsetpoint - Vout;
    term1 = Kp*err + Ki*integrate(err, intStates)
        + Kd*derivative(err, derivStates);
    term2 = Kfeedforward*Vsetpoint/max(Vin,minVin);
    modIndex = max(min(term1 + term2,maxModIndex),minModIndex);

    return modIndex;
}
```

first row of Table 4 shows that regulator(...) is the only function that participates in the application layer, and thus "regulator.c" will be the only file automatically parsed for use in the MIL-DT model. The dependencies of "regulator.c" make up the support layer. See Figure 1.

3.1 Identify Hardware Components to be Modelled

The schematic shown in Figure 2 represents a buck converter with ideal, synchronous switching. No details are given about sensors, filters, communication latency, modulator logic or gate drivers. After including straightforward interpretations of these aspects of the model, the Simulink diagram of the unit under test as shown in Figure 3 is unmodified from the original schematic in Figure 2. The regulation subsystem is shown in Figure 4 along with the modulation logic that is used to drive the switching devices. The logic highly simplified, and there is no dead time between the on time of the upper switch and the on time of the lower switch.

3.2 Automate Parsing or Importing of the Application Layer

In order to demonstrate the "Automated Parsing / Import" portion of the MIL-DT (see Figure 1), a custom import script was written to parse the C code provided in Table 2. This parser creates a Simulink diagram that represents the application Layer software of the MIL-DT. The entry point for the parser is given in Table 5, where the code provides a straightforward interface to the function "parseOneLine.m" shown in Table 6. Various additional functions needed to support parsing are provided in Table 7 and Table 8. The command line interface

Table 3: Embedded Source: Main Loop

main.c

```
// type declarations
typedef integratorStateVars;
typedef derivativeStateVars;

// function headers
double dataIn(double *Vin, double *Iin, double *Vout, double *Iout);
double dataOut(double modIndex);
double regulator(double Vsetpoint, double Vin, double Iin, double Vout,
        double Iout, double ts);
double derivative(double in, derivativeStateVars &states);
double integrator(double in, integratorStateVars &states);

int main() {
    double Vsetpoint, Vin, Iin, Vout, Iout; // input signals
    double modIndex; // output signals
    double ts = 1e-6; // constants

    while (1) {
// receive data from sensors
        dataIn(&Vsetpoint, &Vin, &Iin, &Vout, &Iout);

// compute the modindex
        modIndex = regulator(Vsetpoint, Vin, Iin, Vout, Iout, ts);

// send commands to the gate driver
        dataOut(modIndex);
    }

    return 0;
}
```

Table 4: Partitioning the Code into Application and Support Functions

Function Name	**Application Layer, Support Layer, or Interface**	**Prototyped in**	**Defined in**	**Used in**
regulator	Application	main.c	regulator.c	main.c
main	Support	C standard	main.c	main.c
dataIn	Support	main.c	Not given	main.c
dataOut	Support	main.c	Not given	main.c
derivative	Support	main.c	Not given	regulator.c
integrator	Support	main.c	Not given	regulator.c
max	Support	C standard	C standard	regulator.c
min	Support	C standard	C standard	regulator.c
+(add)	Support	C standard	C standard	regulator.c
- (subtract)	Support	C standard	C standard	regulator.c
* (multiply)	Support	C standard	C standard	regulator.c
/ (divide)	Support	C standard	C standard	regulator.c
= (assignment operator)	Support	C standard	C standard	regulator.c

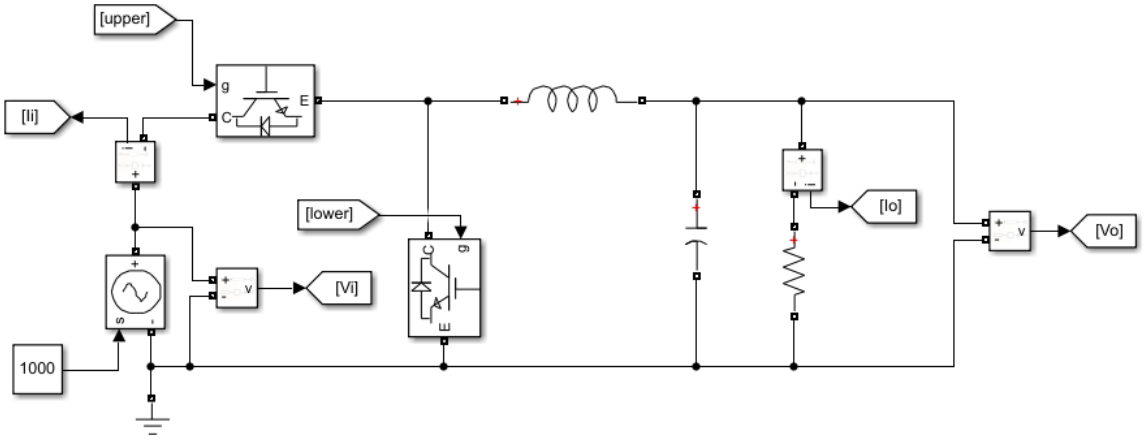

Figure 3: Simulink Diagram of the Buck Converter

of the parsing algorithm is: `parse_C_to_Simulink('regulator.c')`. The output of the conversion is a Simulink diagram shown in Figure 5.

3.3 Code or implement an API between the Application layer and Support Layer

The imported application layer software is represented by the Simulink diagram in Figure 5. Since the parser makes use of built-in Simulink blocks, there is no need to add additional API functionality. Consider though, if custom blocks were needed that were not available in Simulink, for instance: bit packing, encryption, complex data structures, custom lookup tables, or custom discretization of common functions. In cases like this a custom library of Simulink blocks or S-functions might be needed.

3.4 Run Simulations

Simulations were run to estimate the response of voltage regulation against a resistive load of 1 ohm. The commanded output voltage setpoint ramps from 0 to 350 V at a rate of 500 V/s, and experiences an instantaneous step increase from 350 V to 400 V at 2 seconds. The resulting voltage regulation is relatively poor as shown in the blue trace of Figure 6.

The slow convergence of the original regulator provides motivation to find a better alternative. The revised controller has an integral gain increased by a factor of 10 to 0.01, and a feedforward term to provide partial compensation of the nominal modulation index. The benefit of the MIL-DT approach is now evident, as changes can be made directly in the original C file, and re-imported using the scripts that have already been created. Modifying the imported model is not necessary. See Table 2 for the updated contents of "regulator.c". The Simulink diagram that results from the new version of "regulator.c" is given in Figure 7. The modified feedback controller results in significantly better regulation performance, as expected. Note that this was achieved without needed to code any new Simulink files by hand, and without having to compile and link C code against new or existing libraries.

4 Discussion and Conclusions

This paper has discussed Digital Twin simulations as a natural extension to other types of modelling and testing methodologies, including Model-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop. This paper has intended to combine ideas of Digital Twin (i.e. to create a high fidelity digital copy of a real-world artefact) with Model-in-the-Loop simulations (where the goal is to model the behaviour, although perhaps not the implementation details, of a plant being regulated by a feedback system). The resulting methodology called Model in the Loop Digital Twin (MIL-DT) provides a way to create a highly accurate model of the hardware / software interaction in a target system, without needing to directly compile and run the entire codebase in MATLAB / Simulink or some similar toolchain.

The primary drawback to the MIL-DT technique is that it makes extensive use of code parsing, stubbing, and behavioural recoding of existing functions. Note that the original C file in Table 2 only has a handful of lines of functional code, but the MIL-DT model requires a parser that is over a hundred lines long in Table 5, Table 6, Table 7, and Table 8. The parser is, of course, reusable, but the effort to create such scripts cannot be overlooked. As with any high resolution model the benefits of this method must be weighed against the effort to create and maintain import and parsing tools.

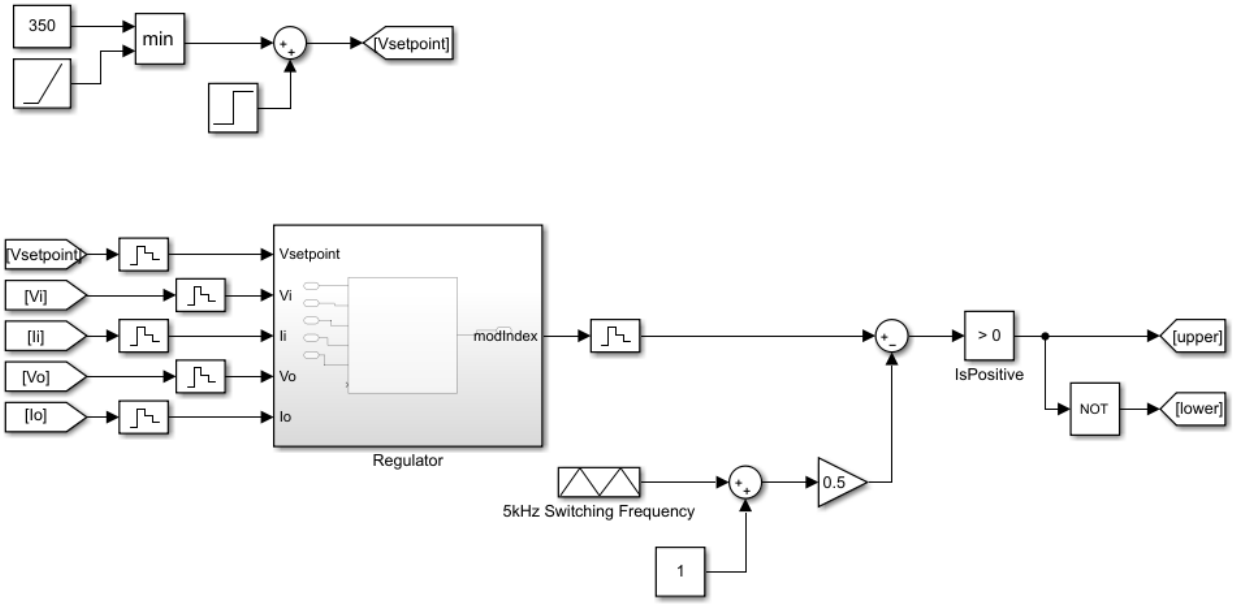

Figure 4: Regulator, Gate Logic, and Setpoint Management Subsystems

Table 5: Matlab Parser to Convert a Simple C file to a Simulink Diagram

parse_C_to_Simulink.m

```
function parse_C_to_Simulink(filename)
    [~,basename,~] = fileparts(filename);

    fid_in = fopen(filename,'r'); % open the C file

    % create and open a new Simulink diagram
    new_system(basename);
    open_system(basename);

    temIndex = 0; % initialize a counter for new variables

    str = fgetl(fid_in); % read a line
    while ischar(str)
        [temIndex] = parseOneLine(basename,str,temIndex);
        str = fgetl(fid_in); % read the next line
    end

    % save the close the Simulink diagram
    save_system(basename);
    close_system(basename);

    fclose(fid_in); % close the C file
end
```

Table 6: Support Function to Parse a Single Line of C Code

parseOneLine.m

```
function [temIndex] = parseOneLine(name,str,temIndex)
    varExp = '[ ]*([a-zA-Z][a-zA-Z0-9]*)[ ]*';
    numericExp = '[ ]*([0-9]+[\\.]?[0-9]*[eE+-]*[0-9]+)[ ]*';

    expr = {};

    expr{end+1} = {sprintf('double %s[\\,\\)]',varExp),'input',1,2};
    expr{end+1} = {sprintf('return %s;',varExp),'output',1,0};
    expr{end+1} = makeAssignment({sprintf('%s',numericExp),'constant',2,1});

    expr{end+1} = makeAssignment(parseFunctionExpression('integrate',2,2));
    expr{end+1} = makeAssignment(parseFunctionExpression('derivative',2,2));
    expr{end+1} = makeAssignment(parseFunctionExpression('max',2,3));
    expr{end+1} = makeAssignment(parseFunctionExpression('min',2,3));

    expr{end+1} = makeAssignment(mathematicalOperator('*','multiply',3));
    expr{end+1} = makeAssignment(mathematicalOperator('/','divide',3));
    expr{end+1} = makeAssignment(mathematicalOperator('+','add',3));
    expr{end+1} = makeAssignment(mathematicalOperator('-','subtract',3));

    expr{end+1} = parseFunctionExpression('integrate',2,2);
    expr{end+1} = parseFunctionExpression('derivative',2,2);
    expr{end+1} = parseFunctionExpression('max',2,3);
    expr{end+1} = parseFunctionExpression('min',2,3);

    expr{end+1} = mathematicalOperator('*','multiply',3);
    expr{end+1} = mathematicalOperator('/','divide',3);
    expr{end+1} = mathematicalOperator('+','add',3);
    expr{end+1} = mathematicalOperator('-','subtract',3);

    for jj=1:length(expr)

        [matchStr,matchTok] = trimregexp(str,expr{jj}{1});

        for ii=1:length(matchStr)
            functionName = expr{jj}{2};
            tokens = matchTok{ii};
            numTokensNeeded = expr{jj}{3};

            formatSpec = makeExpression('makeBlock',numTokensNeeded+2);

            if (expr{jj}{4} == 0)
                argList = {name,functionName,tokens{1:numTokensNeeded}};
                tem = sprintf(formatSpec,argList{:});
                fprintf('%s\n',tem);evalin('base',tem);
                str = '';
            elseif (expr{jj}{4} == 1)
                varName = sprintf('Z%i',temIndex); temIndex = temIndex + 1;
                argList = {name,functionName,varName,tokens{1:numTokensNeeded-1}};
                tem = sprintf(formatSpec,argList{:});
                fprintf('%s\n',tem);evalin('base',tem);
                str = strrep(str,matchStr{ii},varName);
                temIndex = parseOneLine(name,str,temIndex);
                return;
            elseif (expr{jj}{4} == 2)
                if contains(str,name)
                    argList = {name,functionName,tokens{1:numTokensNeeded}};
                    tem = sprintf(formatSpec,argList{:});
                    fprintf('%s\n',tem); evalin('base',tem);
                end
            end
        end
    end
end
```

Table 7: Additional Support Functions used by parseOneLine.m

```
function out = getVarExp()
    out = '[ ]*([a-zA-Z][a-zA-Z0-9]*)[ ]*';
end

function out = makeAssignment(varargin)
    varExp = getVarExp();
    out = varargin{1};
    out{1} = [sprintf('%s[ ]*=',varExp),out{1},'[ ]*[\;\,]'];
    out{4} = 0;
end

function out = makeExpression(functionName,numArgs)
    out = sprintf('%s(',functionName);
    for ii=1:numArgs-1
        out = [out,'''%s'',];
    end
    out = [out,'''%s'');'];
end

function out = mathematicalOperator(operatorSymbol, operatorName, numArgsInMatlab)
    varExp = getVarExp();
    searchExp = sprintf('%s[%s]%s',varExp,operatorSymbol,varExp);

    out = {searchExp,operatorName,numArgsInMatlab,1};
end

function out = parseFunctionExpression(functionName, numArgsInC, numArgsInMatlab)
    varExp = getVarExp();

    searchExp = sprintf('[ ]*%s',functionName);
    if (numArgsInC == 1)
        searchExp = [searchExp,'\(',varExp,'\)'];
    else
        searchExp = [searchExp,'\('];
        for ii=1:numArgsInC-1; searchExp = [searchExp,varExp,','];end
        searchExp = [searchExp,varExp,'\)'];
    end

    out = {searchExp,functionName,numArgsInMatlab,1};
end

function [matchStr,matchTok] = trimregexp(str,searchExpression)
    [matchStr,matchTok] = regexp(str,searchExpression,'match','tokens');
    for ii=1:length(matchStr)
        matchStr{ii} = strtrim(matchStr{ii});
    end
    for ii=1:length(matchTok)
        for jj = 1:length(matchTok{ii})
            matchTok{ii}{jj} = strtrim(matchTok{ii}{jj});
        end
    end
end
```

Table 8: Support Functions Utilizing Simulink for Scripted Block Creation

makeBlock.m

```
function h = makeBlock(diagramName,blockType,blockName,varargin)
    sourceBlocks = varargin;
    makeConnectionFlag = 1; % default

    if strcmp(blockType,'input')
        h = add_block('simulink/Sources/In1',...
            [diagramName,'/',blockName]);
        makeConnectionFlag = 0;
    elseif strcmp(blockType,'output')
        sourceBlocks = {blockName};
        blockName = [blockName,'_out'];
        h = add_block('simulink/Sinks/Out1',...
            [diagramName,'/',blockName]);
    elseif strcmp(blockType,'constant')
        h = add_block('simulink/Sources/Constant',...
            [diagramName,'/',blockName]);
        set(h,'Value',num2str(varargin{1}));
        makeConnectionFlag = 0;
    elseif strcmp(blockType,'add')
        h = add_block('simulink/Math Operations/Add',...
            [diagramName,'/',blockName]);
    elseif strcmp(blockType,'subtract')
        h = add_block('simulink/Math Operations/Add',...
            [diagramName,'/',blockName]);
        set(h,'Inputs','+-');
    elseif strcmp(blockType,'multiply')
        h = add_block('simulink/Math Operations/Product',...
            [diagramName,'/',blockName]);
    elseif strcmp(blockType,'divide')
        h = add_block('simulink/Math Operations/Divide',...
            [diagramName,'/',blockName]);
    elseif strcmp(blockType,'integrate')
        h = add_block('simulink/Continuous/Integrator',...
            [diagramName,'/',blockName]);
    elseif strcmp(blockType,'derivative')
        h = add_block('simulink/Continuous/Derivative',...
            [diagramName,'/',blockName]);
    elseif strcmp(blockType,'min')
        h = add_block('simulink/Quick Insert/Math Operations/Min',...
            [diagramName,'/',blockName]);
    elseif strcmp(blockType,'max')
        h = add_block('simulink/Quick Insert/Math Operations/Max',...
            [diagramName,'/',blockName]);
    else
        error(sprintf('Unknown block: %s',blockType));
    end

    if (makeConnectionFlag)
        for ii=1:length(sourceBlocks)
            makeOneConnection(diagramName,sourceBlocks{ii},1,blockName,ii);
        end
    end

    Simulink.BlockDiagram.arrangeSystem(diagramName)
end

function h = makeOneConnection(diagramName,sourceBlock, ...
                sourcePortNumber,destBlock,destPortNumber)
    h = add_line(diagramName,[sourceBlock,'/',num2str(sourcePortNumber)],...
        [destBlock,'/',num2str(destPortNumber)]);
end
```

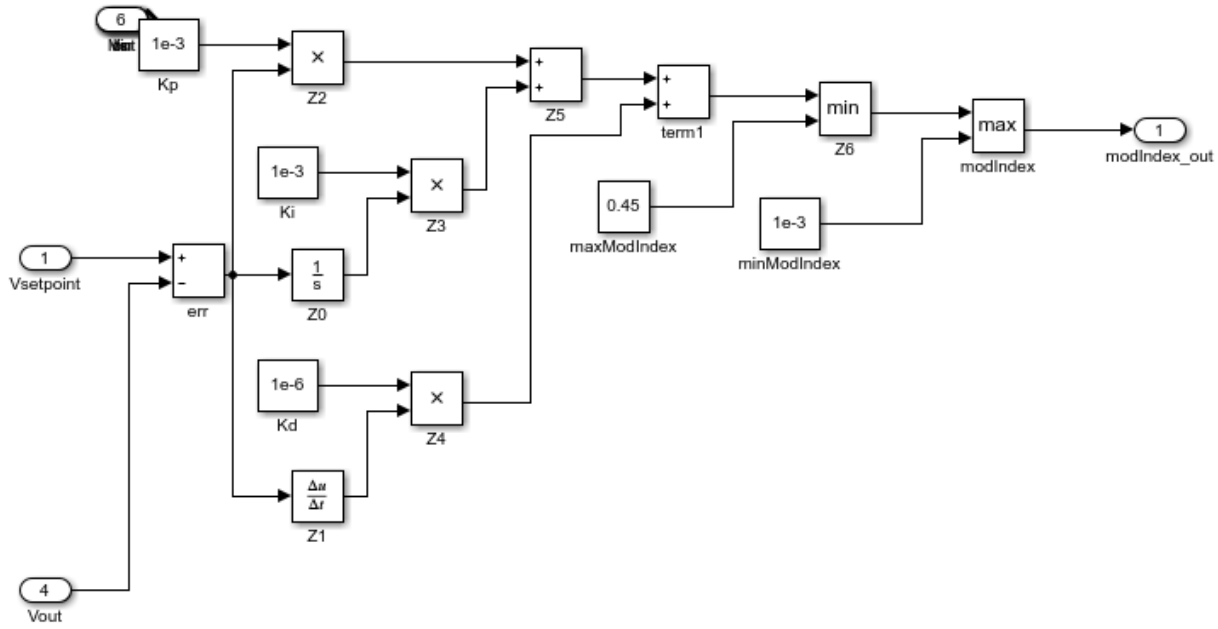

Figure 5: Regulation Algorithm imported as a Simulink Diagram

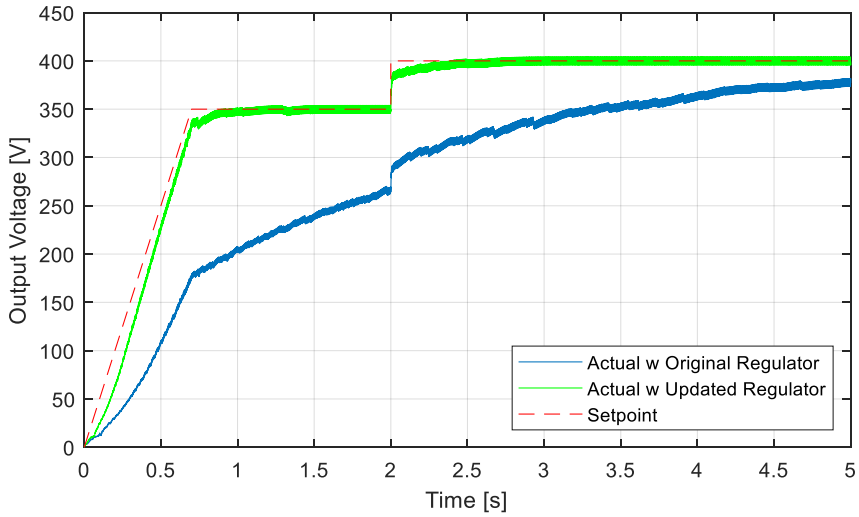

Figure 6: Simulated Voltage Regulation after importing the original C file (in blue) and after importing the modified C file (in green).

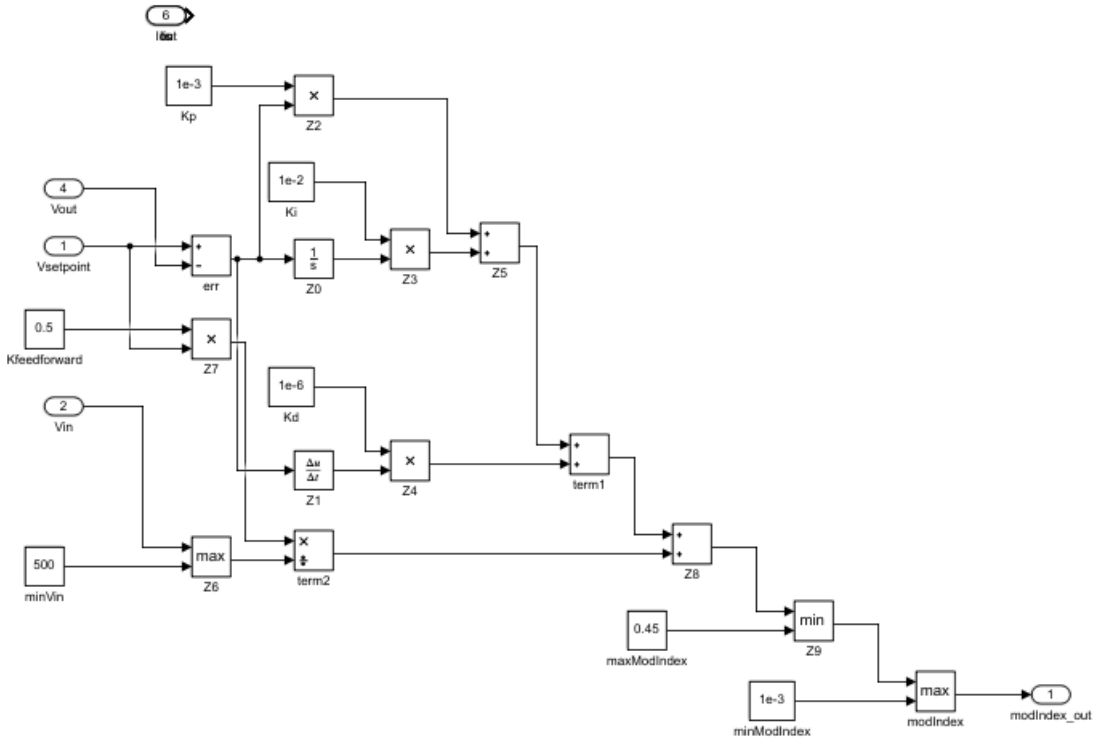

Figure 7: Simulink diagram created by running the import script in Table 5 on the updated source file in Table 2.

The primary strength of MIL-DT is that it can be applied to an existing codebase to create a model without having to change the way the software is developed in the first place. This is important on legacy programs or late stage development programs, where cost to switch from hand coding to auto coding would be prohibitively high. Thus, MIL-DT models can be developed for a wide variety of complex systems so long as the underlying code base can be parsed and translated to a format that is suitable for simulation. Note that the original C file could be modified as shown, also in Table 2, and reimported as a Simulink diagram as shown in Figure 7 without any change to the underlying parser or MIL-DT development process. Overall, once the automatic import techniques are functional, they can be re-used over and over to maintain high accuracy models of hardware / software interaction.

Once a verified model is available it can be used to run any number of automated unit tests, performance verification, or simulation scenarios, all with little to no dependency on actual hardware. The ability to run unit tests and simulation scenarios in a pure software environment enables significantly better scaling and development best practices than in a lab based or prototype based regression testing environment alone.

Even high performance simulations, though, are no substitute for data collected on a physical prototype, and any effective embedded software development process must involve both simulation and testing. By utilizing the scalability of MIL-DT simulations and the reliability of data from physical prototypes, large scale projects could solve problems “at the first available moment”. Errors in data types, event sequencing, and regulation convergence can often be resolved in a digital environment such as MIL-DT, which leaves prototype hardware free for qualification testing, final validation, and root cause debugging of issues that are difficult to simulate.

References

Börcsök, J., Chaaban, W., Schwarz, M.H., Sheng, H., Sheleh, O., Batchuluun, B., 2009. An Automated Software Verification Tool for Model-Based Development of Embedded Systems with Simulink®. 2009 XXII International Symposium on Information, Communication and Automation Technologies , 1–6.

Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W., 2018. Digital Twin in Manufacturing: A Categorical Literature Review and Classification, in: 16th IFAC Symposium on Information Control Problems in Manufacturing (INCOM), Bergamo, Italy.

Krizan, J., Ertl, L., Bradac, M.G., Jasansky, M., Andreev, A., 2014. Automatic code generation from matlab/simulink for critical applications. 2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE) , 1–6.

Kumar, R., Priavrat, P., Dubey, S.A., 2021. A Review on Simulation in Digital Twin for Aerospace, Manufacturing and Robotics. Proceedings of Materials Today 38, 174–178.

Madni, A.M., Erwin, D., Madni, C.C., 2021. Digital Twin-enabled MBSE Testbed for Prototyping and Evaluating Aerospace Systems: Lessons Learned, in: Proceedings of the 2021 IEEE Aerospace Conference.

Magargle, R., Johnson, L., Mandloi, P., Davoudabadi, P., Kesarkar, O., Krishnaswamy, S., Batteh, J., Pitchaikani, A., 2017. A Simulation-Based Digital Twin for Model-Driven Health Monitoring and Predictive Maintenance of an Automotive Braking System, in: Proceedings of the 12th International Modelica Conference, Prague, Czech Republic.

Nibert, J., Herniter, M., Chambers, Z., 2012. Model-based system design for mil, sil, and hil. World Electric Vehicle Journal 5, 1121–1130.

Shafto, M., Conroy, M., Doyle, R., Glaessgen, E., Kemp, C., LeMoigne, J., Wang, L., 2010. Modeling, Simulation, Information Technology and Processing Roadmap.

Shahin, M., Ali Babar, M., Zhu, L., 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access 5, 3909–3943.

Voas, J., Mell, P., Piroumian, V., 2021. Considerations for Digital Twin Technology and Emerging Standards, NIST IR 8356, Draft released: April 16, 2021, Draft retired: April 18, 2022. NIST National Institude of Standards and Technology .

Williams, L., Kudrjavets, G., Nagappan, N., 2009. On the Effectiveness of Unit Test Automation at Microsoft, in: 2009 20th International Symposium on Software Reliability Engineering, pp. 81–89.