An Automated Change Process for Embedded Controller Software of a Full Authority Digital Engine Control System

Lakshmi Vinod Mahesh M. Rao Hem Kapil Shobha S. Prabhu A.N.V. Rao

Scientist, Gas Turbine and Research Establishment,
C V Raman Nagar, Bangalore, India

ABSTRACT
A Full Authority Digital Engine Control (FADEC) system is used in the development and testing of aero-engines and its derivatives at the Gas Turbine Research Establishment. This system incorporates a dual-redundant Digital Electronic Control Unit with embedded software performing control functions. In the development phase of the engine, the control schedules and algorithms are continuously evolving resulting in frequent changes in the control software. Consequently, numerous software versions called builds are generated for different engines. The embedded software, being an extremely critical component of the control system, demands a high degree of reliability in the change management practices. Manual software changes carried out on a large scale are not only error prone but also time consuming thereby leading to slippages in stringent deadlines and entail high cost of correction. Hence, to enhance the reliability and quality of the software, a robust fully automated software change management process has been developed. This process ensures shorter turnaround time and minimizes human errors thereby improving the quality of the safety critical embedded software. This automated change process has been very useful in reducing the development and testing time of the aero-engines and its derivatives.

General Terms
Embedded software, automatic code generation, coding standard, control algorithms

Keywords
FADEC, MATLAB Simulink, MDL, MISRA C, LabVIEW

1. INTRODUCTION
The design and development of Full Authority Digital Engine Control (FADEC) System is one of the critical areas in the development cycle of a gas turbine engine [1]. In the course of development, different control system configurations are being tested on different engines. These changes in configuration require changes in the control system hardware and software. During engine testing, the schedules for fuel flow, compressor variable geometry and exhaust nozzle area along with validation limits for sensors/actuators are established. Control algorithms, gains and time constants of various control loops also need to be fine tuned to meet the overall engine testing requirements. Such a change scenario can precipitate into total chaos if changes are not implemented within stipulated time, reported clearly or controlled in a systematic manner [2]. Change issues become more pronounced and complex in the context of safety and time critical software especially when it is embedded. This calls for a process where changes take place in a systematic and controlled manner [3].

There are three types of frequent changes identified for automation which are discussed in this paper. These changes are automated through an in-house software developed using NI LabVIEW 2011 [4] on a Windows platform. LabVIEW was chosen as the development platform as the learning curve and the implementation time is very short for programmers. There are numerous readymade VIs (Virtual Instruments) provided which can be called directly by the user without any additional coding. Moreover, functions written in C/C++ compiled into Dynamic Link Libraries (DLL) can be ported into LabVIEW framework very easily using facilities of Code In Node (CINs) and Call Library Function nodes [4].

In our case, control law algorithms are represented as MATLAB Simulink blocks which need to be converted to C source code. This code constitutes only a portion of the complete embedded control software. There is an in-house coding standard along with MISRA C 2004 standard to be followed for coding and the auto code generation process has to produce code that adheres to it. This code has to seamlessly integrate with the main application software which does the basic task of scheduling and sequencing. Once integrated, the software is compiled and linked to produce the final executable for the target hardware.

Although automatic code generation is more common these days, the usability of automatically generated C code in safety critical FADEC applications is not completely proven in the industry. Reference [5] and [6] proposes the usage of auto code generated through the Model Based Design (MBD) developing process in safety critical applications. However modeling standards and rules are essential for generation of suitable code. In some applications, it may not be practically possible to impose these restrictions on the MBD in order to obtain safe code. Hence MBD approach will be more useful in rapid prototyping applications rather than in safety critical embedded software. Reference [7] discusses a case study of automatic code generation for safety related applications by comparing various auto-code generation tools in the market. Reference [8] describes a case study of automatic code generation for embedded systems from high-level models using the in-house developed code generator. It emphasizes the need for a code generator that is able to generate code from Matlab Simulink models as no certified code generator for this task is available at the moment. Hence it can be concluded that the code generated through MATLAB Simulink [9] has not only more Lines of Code (LOC) but also not optimum for usage in safety critical applications. Owing to these reasons, the code automatically generated by

Simulink cannot be directly used in our safety critical engine control software in an as-is-where-is condition.

The rest of the paper is organized as follows. Section 2 explains the embedded software change control process and the motivation for automation of the change process. Section 3 explains the implementation of automated code generation. Section 4 presents the case study illustrating the auto conversion of Simulink model to C code. Section 5 evaluates and discusses the benefits of automation. Section 6 concludes the paper with some directions for future work.

2. MOTIVATION

2.1 Frequently Occurring Changes

The embedded software is written in C language and is compiled using a cross compiler for generating the executable object code. This software is functionally divided into two parts, the first part that manages the hardware and its interfaces and the second part that performs the engine control functions. The first part comprises of hardware associated data like scaling coefficients, sensor data validation limits as well as the modules that operate on this data. The second part is comprised of the control laws or algorithms, 2-Dimensional and 3-Dimensional tables representing the various schedules for engine control loops and control gains & time constants. To obtain the optimum performance of the developmental engine under different operating regimes, the control algorithms and these tables need to be changed frequently. The changes discussed in this paper can be broadly classified as follows:

2.1.1 Type I - Constants

This type of change consists of control constants like gains and time constants to fine tune a particular control loop and other constants like hardware coefficients and sensor validation limits. These constants are organized in a C source file as shown in Figure 1.

```
CONSTS.C
const float GAIN1 = 52.65;
const float RNG_LLM = 15.5672;
```

Fig 1: Illustration of constants declaration in software

2.1.2 Type II - Control Schedules

This type of change consists of function tables describing the control schedules for different control loops covering the flight spectrum of the aircraft [1]. These tables are organized in a header file as shown in Figure 2.

```
FUNTAB1.H
const float FUNTAB1[5][6] = {
{       {0.00, 230.0, 240.0, 298.0, 303.0, 315.0},
        {0.30, 0.245, 0.245, 0.245, 0.255, 0.255},
        {0.52, 0.385, 0.385, 0.385, 0.395, 0.395},
        {0.80, 0.543, 0.543, 0.543, 0.645, 0.645},
        {0.12, 0.785, 0.785, 07.85, 0.854, 0.854},
};
```

Fig 2: Illustration of control schedules declaration in software

2.1.3 Type III – Control Laws

This type of change consists of control algorithms that automatically schedule the start-up, acceleration, deceleration, steady state and transient fuel flow by positioning the fuel-metering valve within safe operating limits. A sample Simulink block diagram representing a control algorithm is shown in Figure 3.

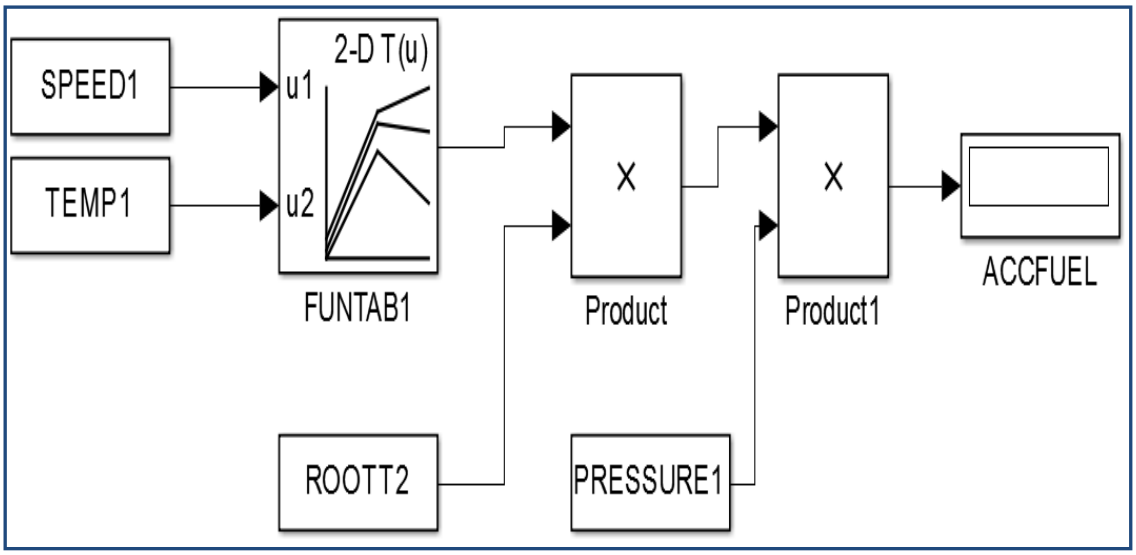


Fig 3: Representation of control law block diagram

2.2 In-house Software Change Process

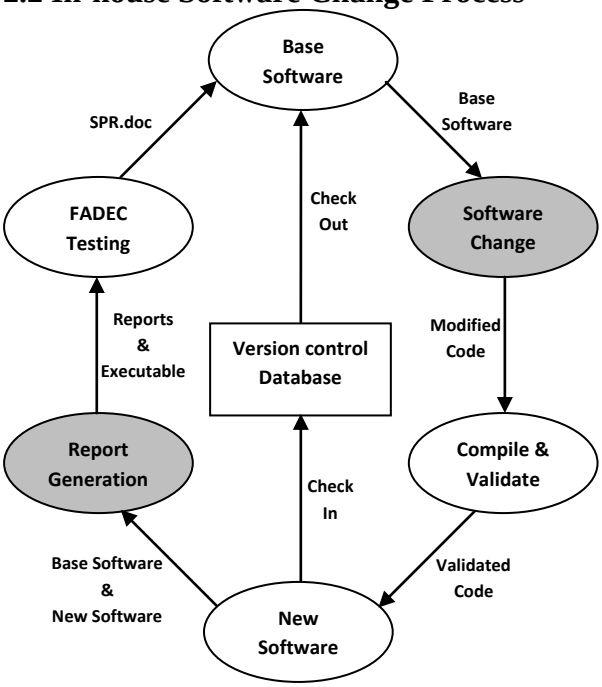


Fig 4: In-house Software Change Control Process

Figure 4 shows the sequence of steps involved in the software change control process. During the control unit testing in test beds/rigs, the software undergoes frequent modifications. Once the software change request is initiated, it is evaluated for its feasibility. The base software mentioned in the Software Problem Report (SPR) is then checked out from the version control database. The software is modified, compiled and the resultant object code is validated to ensure that the intentional changes are correctly carried out. The software is then checked into the database, necessary reports are generated and the final executable is released for engine testing. All these activities are logged into a file to ensure the correctness of the procedures followed and to detect deviations, if any at a later stage. The Total Time to Change (TTTC) follows the equation

$$TTTC = T_{VC} + T_{SC} + T_{RG} + T_{MISC} \quad (1)$$

where T_{VC} is the time taken for version control activities, T_{SC} is the time taken to implement the changes in source code, T_{RG} is the time taken to generate reports for software change management process and T_{MISC} is the time taken to release new software builds and backup actions. For any software change, T_{VC} and T_{MISC} are generally constant. Hence reducing T_{SC} and T_{RG} factors will lead to proportional reduction in the TTTC.

Earlier the software change implementation activity was carried out manually. There was high possibility of human errors especially during data entry of large tables with high precision. Moreover, the control algorithms that are represented as Simulink block diagrams were manually implemented as C source files and compiled for the target platform. Figure 5 reveals the trends in implementation error when the manual software change process was followed for an average of 500 SPRs. Table I presents the implementation time of manual change process per SPR. It is evident from Figure 5 and Table I that the Type III changes in spite of being less frequent contribute to longer implementation time and more errors.

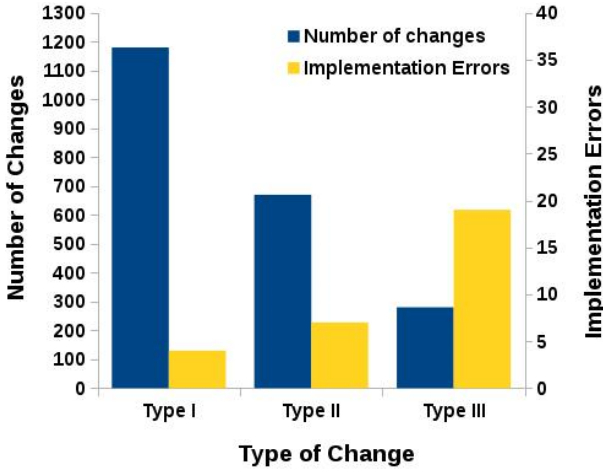


Fig 5: Manual Software Change Process - Number of changes and implementation errors for each Type

Table 1. Implementation Time (in minutes)

Change Type	Source Change Time T_{SC}	Report Generation Time T_{RG}
Type-I	22	5
Type-II	64	5
Type-III	78	10

This situation served as motivation for the design and development of the automated change process, which avoids manual intervention and its resultant human errors, thereby improving software quality.

3. AUTOMATION OF SOFTWARE CHANGE PROCESS

To realize the process automation, LabVIEW 2011 was chosen as the platform for development with separate sub-modules catering to the following types of changes with the aim of reducing T_{SC} and T_{RG} factors.

3.1.1 Type I - Changes

For Type I changes, *CONST_AUTO* sub-module parses the change request document (SPR.doc) for the constants as tokens, performs a multi file search on the project to locate the symbol in the source files and updates the value as mentioned in the change request. The steps for automation process for Type-I changes are shown in Figure 6.

Fig. 6 Automation Process for Type-I Changes (CONST_AUTO)

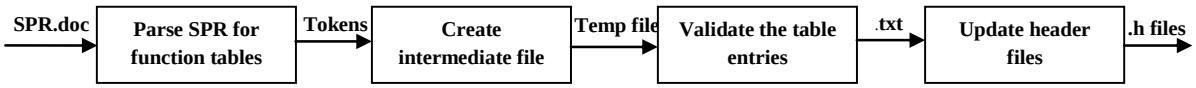


Fig. 7 Automation Process for Type -II Changes (FUNTAB_AUTO)

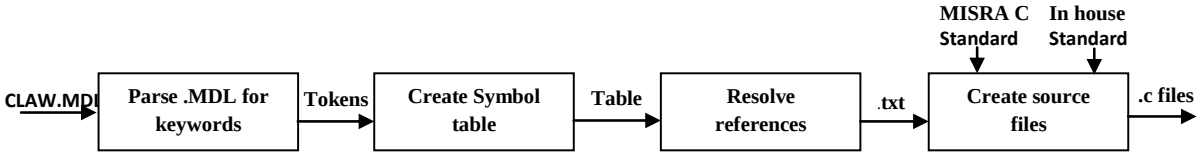


Fig. 8 Automation Process for Type -III Changes (CLAW_AUTO)

3.1.2 Type II - Changes

For Type II changes, *FUNTAB_AUTO* sub-module parses the change request document (SPR.doc) for the function tables as token and creates an intermediate file having table entries. Validation of these table entries are carried out with respect to the following:

- Ascending order of x-axis values for linear interpolation
- Inadvertent reversal of x and y axis values
- Valid float entries
- Table Dimension Change

Once validated, the sub-module automatically changes the required source files and other related files (including header files) of the embedded software. The steps of automation process for Type-II changes are shown in Figure 7.

3.1.3 Type III - Changes

Figure 8 presents the steps for automation process for Type-III changes. Control law algorithms are represented as Simulink model diagrams. In our scenario, these models shall have the following aspects for auto generation of intended and optimized C code:

- The model should only contain pre defined control law blocks (in-house standard)
- All signals required for monitoring should have a name which is represented as a variable in the generated code
- For unnamed signals, local variables will be generated by the automator software as per the in-house variable naming convention

The control law block diagram is represented graphically as shown in Figure 3 and stored in .MDL which is default Simulink model file format as shown in Figure 9. The .MDL file is in plain text format and contains the code for each of the function blocks in the model diagram. For Type III changes, CLAW_AUTO sub-module parses all the required tokens in the text file and generates a symbol table containing all the keywords and tokens. The symbol table references are then resolved to generate the C code for the corresponding model diagram. The code generated is according to the in

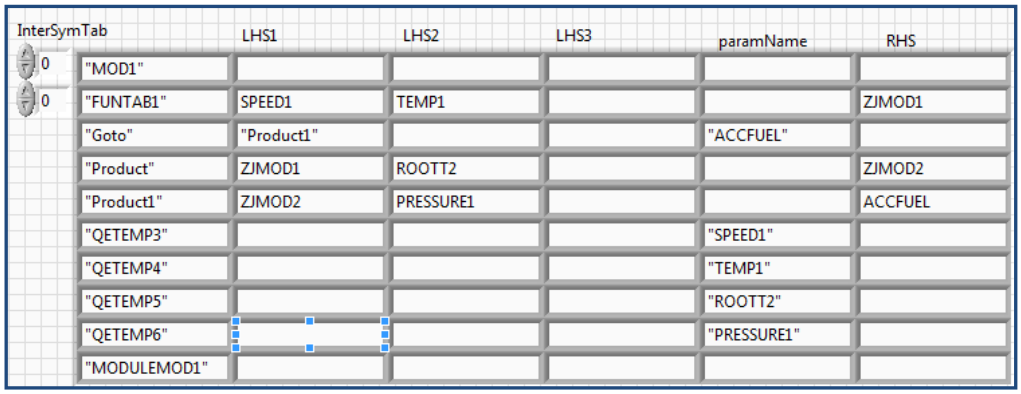
```
Name                    "MOD1"
  Name                    "FUNTAB1"
  Name                    "Goto"
  GotoTag                 "ACCFUEL"
  Name                    "Product"
  Name                    "Product1"
  Name                    "QETEMP3"
  GotoTag                 "SPEED1"
  Name                    "QETEMP4"
  GotoTag                 "TEMP1"
  Name                    "QETEMP5"
  GotoTag                 "ROOTT2"
  Name                    "QETEMP6"
  GotoTag                 "PRESSURE1"
  SrcBlock                "QETEMP6"
  DstBlock                "Product1"
  DstPort                 2
  SrcBlock                "QETEMP5"
  DstBlock                "Product"
  DstPort                 2
  SrcBlock                "Product1"
  DstBlock                "Goto"
  DstPort                 1
  SrcBlock                "Product"
  DstBlock                "Product1"
  DstPort                 1
  SrcBlock                "FUNTAB1"
  DstBlock                "Product"
  DstPort                 1
  SrcBlock                "QETEMP3"
  DstBlock                "FUNTAB1"
  DstPort                 1
  SrcBlock                "QETEMP4"
  DstBlock                "FUNTAB1"
  DstPort                 2
  Name                    "MODULE MOD1"
```

Fig 9: MATLAB Simulink model file format

house coding standards for variable name conventions and MISRA C 2004 [10] standard for safe subset of C language. All the associated header files are changed by this sub-module assuring that all interdependencies within the main application framework are resolved.

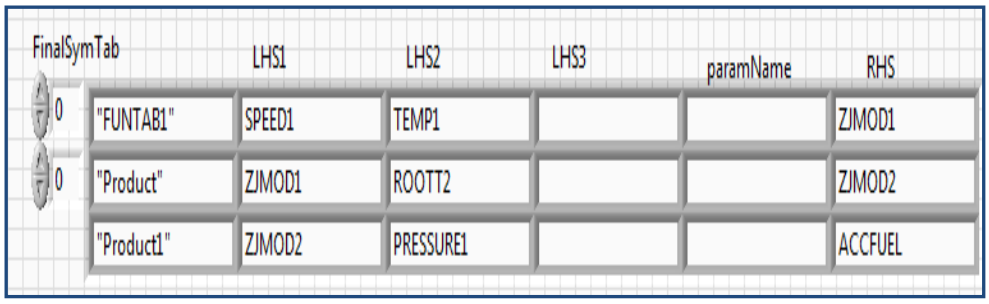
3.1.4 Report Generation:

Earlier when the process was manually carried out, the base and new software builds used to be compared and the differences observed highlighted in the final report. This activity was not only time consuming, but also failed to reflect



InterSymTab	LHS1	LHS2	LHS3	paramName	RHS
0 "MOD1"					
0 "FUNTAB1"	SPEED1	TEMP1			ZJMOD1
"Goto"	"Product1"			"ACCFUEL"	
"Product"	ZJMOD1	ROOTT2			ZJMOD2
"Product1"	ZJMOD2	PRESSURE1			ACCFUEL
"QETEMP3"				"SPEED1"	
"QETEMP4"				"TEMP1"	
"QETEMP5"				"ROOTT2"	
"QETEMP6"				"PRESSURE1"	
"MODULEMOD1"					

Fig 10: Illustration of intermediate symbol table



FinalSymTab	LHS1	LHS2	LHS3	paramName	RHS
0 "FUNTAB1"	SPEED1	TEMP1			ZJMOD1
0 "Product"	ZJMOD1	ROOTT2			ZJMOD2
"Product1"	ZJMOD2	PRESSURE1			ACCFUEL

Fig 11: Illustration of final symbol table

```
void MOD1(void)
{
   float ZJMOD1 = 0;
   float ZJMOD2 = 0;

   ZJMOD1 = FUN2D(FUNTAB1, FUNTAB1_ROWS - 1, FUNTAB1_COLUMNS - 1, SPEED1, TEMP1);
   ZJMOD2 = MUL(ZJMOD1,ROOTT2);
   ACCFUEL= MUL(ZJMOD2,PRESSURE1);

}
```

Fig. 12 Illustration of generated C source code

all changes indicated in the change request. The automatic report generation sub-module *REPORT_AUTO* generates the report based on the parsed information from the change request document, thus reducing the T_{RG} factor. All the changes that are carried out are recorded by this module and the report is generated automatically without missing any details.

4. CASE STUDY

The case study demonstrates a typical Type III change represented by the block diagram shown in Figure 3. Each block diagram is represented as a single C function in the code. The text representation of the .MDL file is shown in Figure 9. This file is parsed for tokens such as *Goto*, *Product* and *Product1* and the parameters are linked to each other through keywords such as *SrcBlock*, *DstBlock* and *DstPort*. Using this information, an intermediate symbol table is formed as shown in the Figure 10.

The symbol table references are resolved and the final symbol table is obtained as shown in the Figure 11. The sub-module CLAW_AUTO generates the C code equivalent of the block diagram from the final symbol table as shown in Figure 12.

Moreover, all the necessary local variables are automatically declared as per the coding standard for each function. Likewise all the block diagrams are converted into C modules and the global variables required to be modified are analyzed and updated accordingly. The C modules are updated into appropriate source files as per coding standards and the software is ready for compilation without any manual intervention.

5. EVALUATION AND DISCUSSION

The automation software was developed in NI LabVIEW2011 and the modules were integrated into a single framework catering to any type of changes as explained in the previous sections. Different models of control algorithms for FADEC were designed and verified by this software. Development process consisted of much iteration as the keywords parsed from the model file were not consistent. Once the modeling guidelines were finalized, the software was able to generate consistent code. The generated code was verified and deployed to the embedded target hardware. The time taken for different type of changes, both manually and through automation is compared and presented in Figure 13 for an average of 100 SPRs.

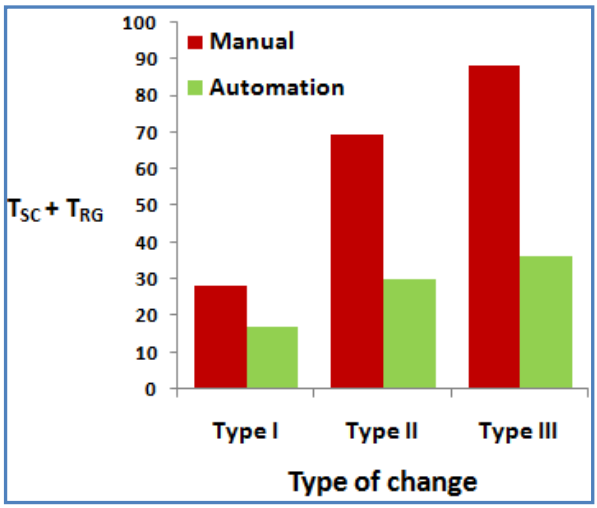


Fig 13: Comparison of turnaround time - Manual and Automation

From Figure 13, it is evident that there is a drastic reduction in the turnaround time because of automation. There is 44% reduction for Type I changes, 57% for Type II and 60% for Type III changes. On an average there is ≈50% of time reduction attained through automation. In addition, it reduces the implementation errors for all type of changes as shown in Figure 14.

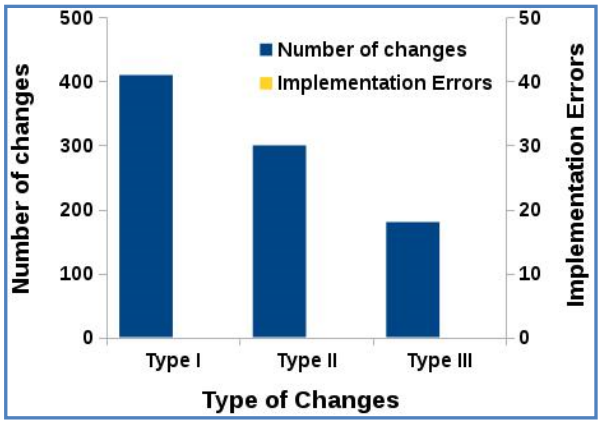


Fig. 14 Automated Software Change Process - Number of changes and implementation errors for each Type

6. CONCLUSION AND FUTURE SCOPE

The automation process explained in this paper enables systematic implementation of software change management procedure which is of paramount importance in organizations involved in developmental projects. It is highly useful for software products that warrant high degrees of reliability and demand frequent accommodation of changes at the same time. Evolving practices analogous to the ones discussed above to ensure correctness of change implementation could benefit these organizations.

Currently this automation software caters to .MDL model format of MATLAB Simulink. In future, the same paradigm can be extended to model formats other than .MDL and also to other model based development tools.

7. ACKNOWLEDGMENTS

The authors are grateful to Director, Gas Turbine Research Establishment, DRDO who gave permission to publish this paper. The authors also thank those associated with the preparation of manuscript for this paper.

8. REFERENCES

[1] B. Githanjali, P. Shobha, K. Ramprasad, and K. Venkataraju, "Full Authority digital engine controller for marine gas turbine engine," in ASME Turbo Expo 2006: Power for Land, Sea, and Air, May 2006, pp. 611–618.

[2] P. Jalote, An Integrated Approach to Software Engineering, 2nd ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[3] R. S. Pressman, Software Engineering: A Practitioner's Approach, 5th ed. McGraw-Hill Higher Education, 2001.

[4] G. W. Johnson, LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control, 2nd ed. McGraw-Hill School Education Group, 1997.

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

[6] Prosvirin, D.A.; Kharchenko, V.P., "Model-based solution and software engineering environment for UAV critical onboard applications," in Actual Problems of Unmanned Aerial Vehicles Developments (APUAVD), 2015 IEEE International Conference , vol., no., pp.312-315,13-15 Oct. 2015 doi: 10.1109/APUAVD.2015.7346629

[7] D. P. Gluch and A. J. Kornecki, "Automated code generation for safety related applications: a case study," in Proceedings of the International Multiconference on Computer Science and Information Technology, 2006, pp. 383–391.

[8] A. Riid, J. Preden, R. Pahtma, R. Serg, T. Lints. Automatic Code Generation for Embedded Systems from High-Level Models Electronics and Electrical Engineering. – Kaunas: Technologija, 2009. – No. 7(95), – P. 33–36.

[9] www.mathworks.in/help/pdf_doc/simulink/slref.pdf

[10] D. Ward, "MISRA standards for automotive software" in The 2nd IEE Conference on Automotive Electronics, March 2006, pp. 5–18.