2012-01-0030

Promoting the Robustness of Calibration-Based Computations

Jay Abraham
MathWorks Inc.

ABSTRACT

Calibration parameters are extensively used in complex automotive Engine Control Units (ECUs), including ECUs for the engine, transmission, Anti-lock Braking System (ABS), and Electronic Stability Control (ESC). Calibration engineers can set the exact values of calibration parameters for a given software application after the ECU software is built. Such parameters also enable a single set of software to control multiple hardware variants, for example 4-cylinder and 6-cylinder engine variants, or turbo and non-turbo variants. In an ECU, there are often hundreds and sometimes tens of thousands of calibration parameters, some of which are multidimensional tables. With this level of complexity, ensuring that the ECU software using the values from these tables will not encounter an overflow operation, divide by zero condition, or illegal memory access run-time error can be a significant challenge. Due to the connection between hardware and software, such errors could potentially cause hardware damage or unexpected behavior, which in turn can lead to end-user safety concerns

With traditional testing it is impossible to exhaustively test such complex software systems, comprised of both calibration parameters and code, to prove that the software is free of run-time errors. Verification based on formal methods may provide a means by which it may be possible to learn more about the quality of the software from a run-time perspective. With formal methods, it is possible to exhaustively verify the software, even software with sophisticated calibration parameters. Using formal methods, engineers can specify the full range of data in the calibration tables and exhaustively verify the software, rather than testing with a limited range of data in the tables.

INTRODUCTION

To meet market and regulatory requirements for engines that are optimally balanced for performance, emissions, and fuel economy, automotive engineers rely on ECU software that can contain tens of thousands of calibration parameters. The calibration parameters are critical data that is consumed by sophisticated algorithms implemented in embedded software. The algorithms may contain operations in fixed and floating point arithmetic that include add, subtract, multiply, divide, and square root among others. These algorithms may directly control various modes of vehicle operation. In the engine, for example, the algorithms may control modes such as fuel economy, performance, limp home, cold start, and run without coolant. For safety, reliability, emissions, and other factors, it is important that the software does not select an incorrect mode as a result of erroneous calibration data or some other failure with the algorithm.

The embedded software that uses the calibration data is frequently implemented in the C programming language. Often generated from a higher level graphical language, this code may rely on the use of fixed size arrays for data storage and variable manipulation. It is impossible to exhaustively verify the software for all types of multidimensional calibration parameters. Therefore, unless adequate protection is built into the embedded software, it is possible that the software may fail with an overflow, divide by zero, square root of a negative number, invalid array access index, or similar run-time error. When reusing an existing library with a new set of calibration data for which it was not originally designed, engineers must ensure that the algorithms in the library can indeed handle the full range of calibration data without encountering run-time errors. Calibration engineers may develop algorithms without full knowledge of the ECU microprocessor on which they will be implemented. This could also cause inadvertent overflow and underflow conditions (due to bit size limitations of the software and/or microprocessor). Formal methods may provide a solution to these dilemmas.

Formal methods apply theoretical computer science fundamentals to solve difficult problems in software, such as proving that the software will not fail with a run-time error. Using formal methods it is possible to identify potential problems in computation that depend on calibration data. Formal methods can also be used to prove that the computation is robust in software that is dependent on calibration tables. Ultimately, formal methods can help confirm ensure that the software and underlying algorithms are robust and that they can handle the full range of calibration data.

WORKING WITH CALIBRATION DATA

Figure 1 shows calibration data being used by an ECU that controls hardware (for example, an engine throttle). The algorithm is implemented in embedded ECU software that runs on a microprocessor. The software may be handwritten or automatically generated from models, such as those implemented in Simulink® and Stateflow®.

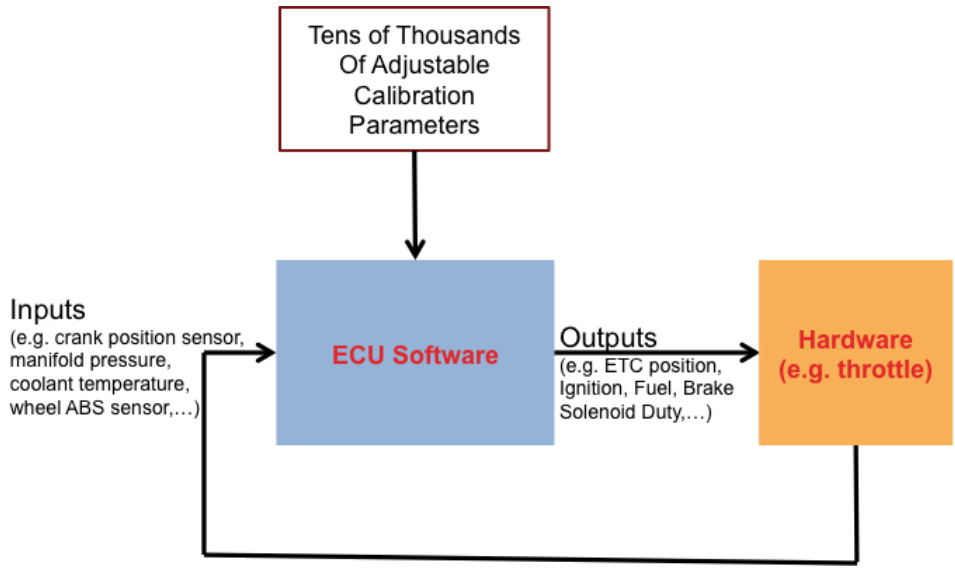

Figure 1 – Use of calibration data with an ECU and hardware

Figure 2 shows the modern process by which automotive engineers develop calibration parameters. The process starts with Design of Experiments (DOE), followed by modeling, physical testing, and simulation. In practice, calibration engineers may tweak calibration data based on driving the vehicle on a test track to optimize the controller.

Figure 2 – Calibration data generation

VERIFICATION CONCERNS

When seeking to check the robust operation of ECUs with calibration-based algorithms, engineers need to assess:

1. Errors due to arithmetic issues, such as divide by zero or taking the square root of a negative number.
2. Errors that stem from accessing memory with pointers. The use of two-stage lookups, for example, in which a value from one table is used as index into a second table, can lead to such errors.

3. Errors that result from reusing an algorithm from a library that does not work correctly with new calibration data or a particular engine variant.
4. Overflow errors caused by a calibration range that is wider than permitted by variables used in an algorithm.
5. And other concerns relating to the robust operation of the ECU

When calibration data is static, it is possible to verify manually that the software will perform in a robust manner, provided that the other inputs to the software (such as temperature, engine speed, and so on) are bounded by constraints. For example, consider the following fragment of code written in C. This code contains data in a static array named *calibration_table_5x4*. With reasonable review and testing efforts, it can be shown that this code will function without any run-time errors (including overflow, divide by zero, and so on).

int16 calibration_table_5x4[20] ***{ 120, 240, 63, 42 … /* fixed calibration data */***

… computation performed on table (i.e. with add/subtract/multiply/etc. operations) …

return (int16) (computed_value + 127);

However, assuming that the calibration data will always remain the same is an unreasonable expectation--especially if the same ECU software will be reused with different engine variants. To exhaustively confirm that this code is robust, the *calibration_table_5x4[]* array must be treated as full-range variables. Under this condition, would it be possible to prove that the computations performed and the returned value will not result in a run-time error? In theory, performing code reviews and executing the right set of test cases may catch every defect, no matter the type. In practice, however, the challenge is the amount of time it takes to thoroughly review code and apply enough of the right tests to find all the errors. Even for the simplest operations, such as adding two 32-bit integer inputs, one would have to spend hundreds of years to complete exhaustive testing, which is not realistic. Using these techniques, it may not be possible to exhaustively show that software that uses calibration data is robust and run-time error free. However, using formal methods it may be possible to prove or disprove the existence of problems or potential problems in the code. For example:

int16 calibration_table_5x4[20]; ***/* calibration data not specified – assumed full range */***

… computation performed on table (i.e. with add/subtract/multiply/etc. operations)…

return (int16) (computed_value + 127);

In the code fragment above, the *calibration_table_5X4* array can now take the full range of values for all of the 20 elements in the table. Using static code analysis based on formal methods it is possible to identify potential run-time errors in the computation and return value. It is also possible to formally prove that the computation is robust. In other words, formal analysis can show that the code can handle the full range of calibration data.

INTRODUCTION TO FORMAL METHODS

Applying formal methods to the analysis of code gives engineers insight into their design and code and confidence that they are robust. To better understand formal methods, consider the following example. Without the aid of a calculator, compute the result of the following multiplication problem within three seconds:

$$-4586 \times 34985 \times 2389 = ?$$

Although computing the answer to the problem by hand will likely take you longer than three seconds, you can quickly apply the rules of multiplication to determine that the result will be a negative number. Determining the sign of this computation is an application of a specific branch of formal methods known as abstract interpretation. The technique enables you to know precisely some properties of the final result, such as the sign, without having to multiply the integers fully. You also know from applying the rules of multiplication that the result will never be a positive number or zero for this computation.

The concept of soundness is important in context of a discussion on abstract interpretation. Soundness means that when assertions are made about a property, those assertions are proven to be correct. The results from abstract interpretation are considered sound because it can be mathematically proven with structural induction (a proof method for mathematical logic)

that abstraction will predict the correct outcome. When applied to software programs, abstract interpretation can be used to prove certain properties of software, for example, that the software will not exhibit run-time errors such as overflow, divide by zero, and illegally dereferenced pointers, among others.

APPLYING FORMAL METHODS TO SOFTWARE VERIFICATION

Static code analysis (also known as static analysis) is a software verification activity in which source code is analyzed for quality and reliability. Metrics produced by static code analysis provide a means by which software quality can be measured and improved. In contrast to other verification techniques, static code analysis can be done without executing the program or developing test cases, and therefore is relatively easy to automate.

Basic static code analysis techniques include verifying compliance to code standards such as MISRA-C/C++ and generating code quality metrics, such as counting the number of lines of code, determining comment density, and assessing code complexity. More sophisticated techniques couple static code analysis with formal methods such as abstract interpretation as described above. The combination of static code analysis and formal methods enables engineers to detect elusive run-time errors and to prove the absence of certain run-time errors in the software. This approach enables variable ranges to be accurately determined and every potential failure point in the code is identified as proven to fail, proven not to fail, may never execute (dead code), or unproven. Tools that support this technique investigate all possible behaviors of a program–that is, all possible combinations of values–in a single pass to determine how and under what conditions the program may exhibit certain classes of defects. For example, Polyspace® code verification products analyze C, C++, or Ada source code to determine where certain run-time errors could occur and then, with color coding, indicate the status of each element in the code (see Figure 3). Tool tips show the range of local and global variables. In Figure 3, the variable "*i*" in the *for* loop can be observed to have an assignment range from 1 .. 100 after the increment (that is, after *i*++ is executed).

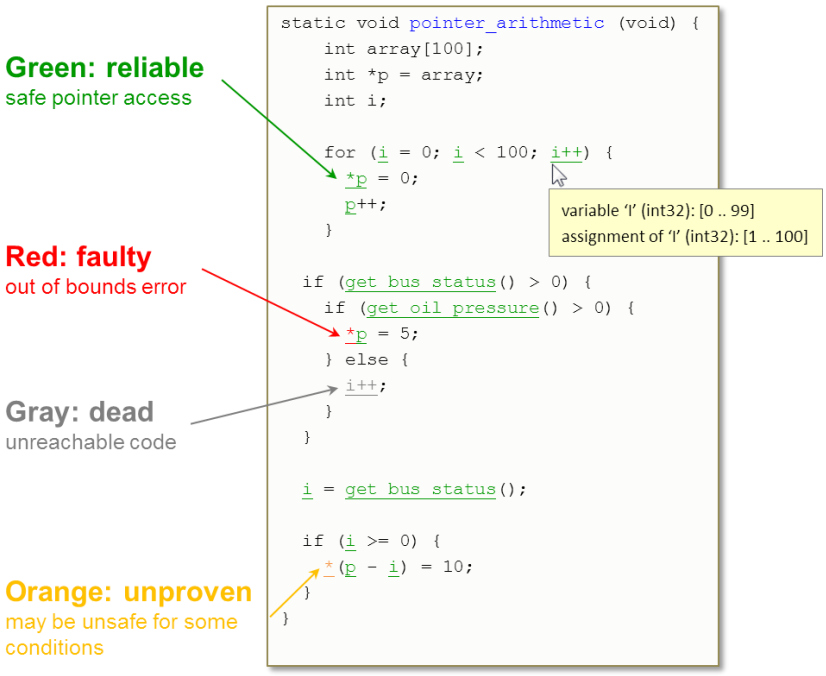

Figure 3 – Polyspace color-coding indicates the status of each element in the code.

Code that Polyspace marks in green is free of certain run-time errors. Code that contains run-time errors is marked as red, code that may contain run-time errors for some conditions is marked as orange, and code that is unreachable is marked as gray. By identifying code that is proven to be run-time error-free, engineers can free more time for fixing identified issues.

EXAMPLES

(1) VERIFYING FOR OVERFLOW

Figure 4 shows a code example for a function called *command_strategy()*. It consists of an algorithm that is dependent on two inputs (*x* and *y*) with calibration data contained in an array *a[]* with 30 elements. Note that the code performs shifts, additions, and subtractions, and it accesses the array with pointer arithmetic.

```
/*  Calibration data for the command strategy (stored in a
global array a) */

   int16_T a[30] = {      120,  240,  360,  480,  600,  720,

                          240,  480,  720,  960,  200,  440,

                          360,  720,   80,  440,  800,  160,

                          480,  960,  440,  920,  400,  880,

                          600,  200,  800,  400,    0,  600};

int16_T command_strategy(uint16_T x, uint16_T y)

{

   /* Intermediate variable declarations  */

   int16_T i, tmp, return_val, *p = a;

          uint16_T tmp_x, tmp_y;

          tmp_x = x;

          tmp_y = y;

  /*  Scale data  */

   for(i = 0; i < 30; i++)

   {

      *p = a[i]>>2;

      p++;

   }
```

```
   /* Scale and saturate inputs*/

   tmp_x = (tmp_x >> 11) - 2; if (tmp_x > 29)  {tmp_x = 29;}

   tmp_y = (tmp_y >> 12);


   /* Compute return value  */

   return_val = a[tmp_x] - a[tmp_y];

   if (return_val < 3)

   {

      tmp = *(p-5) + 5;

      return_val = tmp + 10;

      return (return_val);

   }

   else {

      return (return_val + 10);

   }

}
```

Figure 4 – Source code for calibration example.

By using Polyspace to analyze this code, engineers can identify all of the locations where the software could suffer from certain run-time errors. In this code example, Polyspace identifies 41 such places. Some of these potential errors are:

- Uninitialized variable for *tmp_x = x;*
- Overflow on *(return_val + 10)*
- Pointer dereference for **(p-5)*, which maybe out of bounds
- Out of bound array index check on *a[tmp_x]*
- Pointer initialization for **p = ...*
- Scalar shift is potentially out of bounds for *a[i]>>2*

More importantly, the analysis performed by Polyspace, using the formal methods based technique known as abstract interpretation, shows by proof that the function *command_strategy()* is robust and that Polyspace has not detected run-time errors at any of these 41 points. These are colored in green by Polyspace as shown in Figure 5.

```
/*  Calibration data for the command strategy (stored in a global
int16_T a[30] = {120,   240,   360,   480,   600,   720,
                 240,   480,   720,   960,   200,   440,
                 360,   720,    80,   440,   800,   160,
                 480,   960,   440,   920,   400,   880,
                 600,   200,   800,   400,     0,   600};

int16_T command_strategy(uint16_T x, uint16_T y)
{
    /*
     * Intermediate variable declarations
     */
    int16_T i, tmp, *p = a, return_val;
        uint16_T tmp_x, tmp_y;

        tmp_x = x;
        tmp_y = y;

   /*
     * Scale the data
     */
    for(i = 0; i < 30; i++)
    {
        *p = a[i]>>2;
        p++;
    }
```

```
    /*
     * Scale and saturate inputs
     */
    tmp_x = (tmp_x >> 11) - 2; if (tmp_x > 29)  {tmp_x = 29;}
    tmp_y = (tmp_y >> 12);

    /*
     * Compute return value
     */
    return_val = a[tmp_x] - a[tmp_y];
    if (return_val < 3)
    {
        tmp = *(p-5) + 5;
        return_val = tmp + 10;
        return (return_val);
    }
    else {
        return (return_val + 10);
    }
}
```

Figure 5 – Run-time analysis results.

However, this claim can only be made for the current calibration data contained in the array *a[]*. To determine if this code is completely robust and that it can handle any type of calibration data, it must be assumed that the calibration data is full range; that is, all 30 elements in the array *a[]* must be allowed to assume the full range of values from -32768 to 32767. This verification can be performed by simply modifying the code for the array declaration to the following:

*int16_T a[30]; **/* calibration data not specified – assumed full range */***

With this modification, Polyspace will assume all elements of the array *a[]* are full range (that is, -32768 .. 32767). With this assumption, Polyspace identifies several places where an overflow may occur in the code. For example for the statement *return_val = a[tmp_x] - a[tmp_y]*. The subtraction produces a value that has a range of -65535 .. 65535, which is the result of subtracting two numbers in the range of -32768 .. 32767. Since *return_val* is of data type *int16_T* this could cause an overflow run-time error. Figure 6 shows the result from Polyspace along with the range information displayed from this computation.

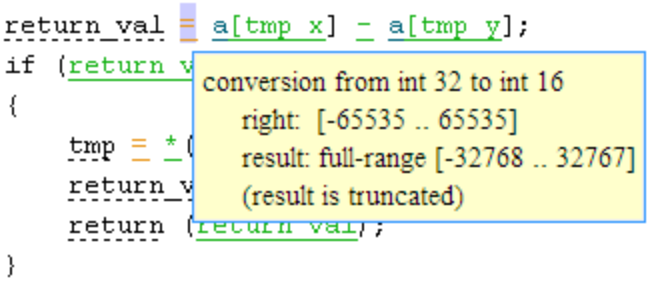

Figure 6 – Polyspace identifying potential overflow condition.

To make this code more robust, it would be necessary to change the data type of *return_val* to *int32_T* so that there is sufficient space to save the larger results. After this change is made and the code re-analyzed with Polyspace, the overflow error is no longer a possibility and is therefore not reported. The *"="* operation is no longer colored in orange and the result now fits in the larger *return_val* variable as shown in Figure 7.

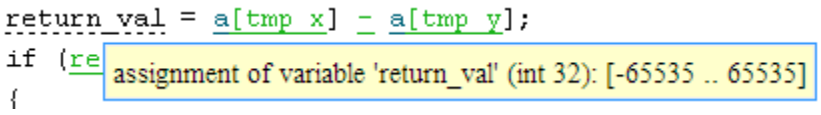

Figure 7 – Polyspace no longer flags the overflow condition.

(2) USING RANGE INFORMATION

Example (1) showed how one can develop a robust algorithm that will not fail due to run-time errors even if the calibration data is full range. There may be some situations in which it is desirable to specify a narrow range for calibration data. The expectation is that the computations performed will not fail with a narrow range specification. This is possible with tools like Polyspace. Taking the same code example used previously with the original *return_val* data type, one can specify in the tool that the calibration data for the table in array *a[]* is to be in the range of 0 .. 1000. As shown in Figure 8, with these limits, the variable *return_val* with data type *int16_T* will not overflow. Notice also that the computed range of *return_val* is now -1000 .. 1000.

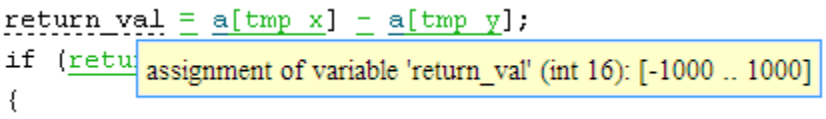

Figure 8 – Polyspace results with range limits applied.

(2) CHECKING THAT MODE SPECIFICATIONS STAY WITHIN RANGE

Variables within an algorithm may be used to track specific modes; for example, fuel economy mode, performance, limp home, cold start, run without coolant, and so on. Using tools like Polyspace, it is possible to check that the wrong mode does not activate due to a problem with the calibration data or algorithm. For example, if the code contained several *if-else-if* statements or a *case* statement that would execute specific operations based on a mode, Polyspace will be able to determine if all of the statements will be activated with specific calibration data, a specified range data, or the full range conditions. If some of the case statements are colored in gray by Polyspace, then has been shown that those conditions will not arise with the use of that specific calibration data or range.

For example, as shown in Figure 9, the *engine_mode* variable is set based on the variable *return_val*. In this example, *return_val* is in the range of -8177 .. 8206. However, as implemented, the code activates COLD_START mode only when this variable is < -9000. Since this is not possible, this particular ECU will never activate the engine in COLD_START mode. The result can be seen in Figure 9. Note the grayed out curly bracket "*{*" and that COLD_START is not colored in green, which is indicative of dead code.

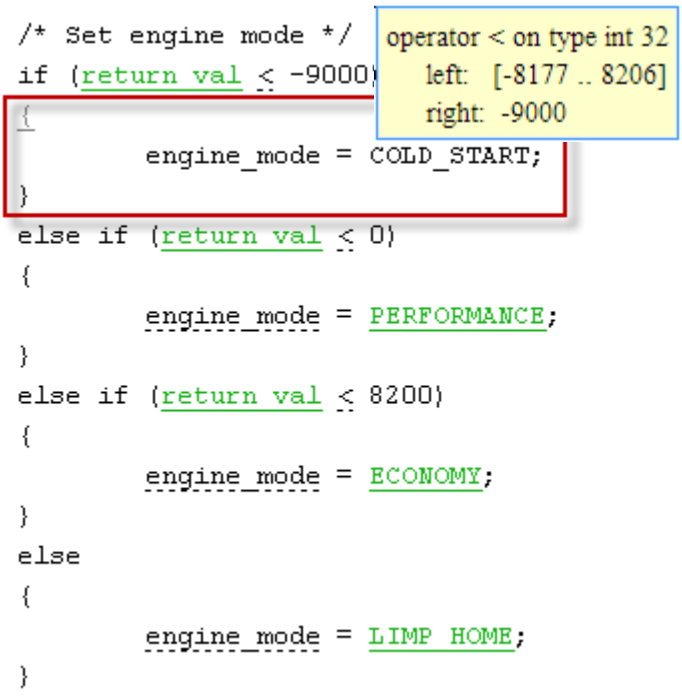

Figure 9 – Polyspace identifying dead code.

SUMMARY/CONCLUSIONS

The pressure to improve engine efficiency coupled with desire to minimize cost of manufacturing is fueling the increased use of complex multidimensional calibration parameters in automotive ECUs. With this level of complexity, ensuring that the ECU software, which uses the values from these tables, does not run into an overflow operation, divide by zero condition, or illegal memory access can be a challenge. With traditional testing it is impossible to exhaustively test these complex data and code software systems to prove that the software is run-time error free. With formal methods, however, it is possible to exhaustively verify the software. For calibration purposes, rather than testing with a limited range of data in the tables, engineers can specify a full range of data in the calibration tables and exhaustively test the software. When coupled with static code analysis, formal methods can show if and where the software will potentially fail with a run-time error. By identifying where run-time errors are not present in code, engineers can focus their efforts on fixing identified issues in their software algorithms that are dependent on calibration data.

REFERENCES

1. Kampelmühler, Paulitsch, Gschweitl, “Automatic ECU-Calibration - An Alternative to Conventional Methods”, SAE. 1993
2. Akella, “Automotive Embedded Software Verification and Validation Strategies”, Emmeskay. 2009
3. Pan, “Dependable Embedded Systems”, Software Testing. 1999
4. Cousot and Cousot “Abstract Interpretation Based Formal Methods and Future Challenges”, Informatics. 10 Years Back. 10 Years Ahead, 2001
5. Deutsch, “Static Verification of Dynamic Properties, SIGAda, 2003
6. www.mathworks.com/products/polyspace

DEFINITIONS/ABBREVIATIONS

ECU	Engine Control Unit
ABS	Anti-lock Braking System
ESC	Electronic Stability Control
ETC	Electronic Throttle Control