Test-Driven Development Methodology for Complex Algorithms

Efficient development of computation intensive algorithms

Anto Michael, Llarina Lobo Palacios, Sebastian Zuther;
Valeo Switches and Sensors GmbH, Germany

Traditionally, embedded software written for the automotive industry typically used to get information from sensors and control the actuators in the vehicle. Since the beginning of this century, the focus has slowly switched towards various levels of assistance to the person at the steering of the vehicle. It started with the introduction of obstacle warning systems at low speeds. Then vehicles came that could detect and park automatically into parking spots under the observation of the driver. Lane change assist and automatic cruise control systems are becoming more and more common in vehicles. The world is now advancing towards automated driving platforms. The most important part of the autonomous driving is to perceive the environment around the vehicle. The perception layer uses information from sensors like ultrasonic, camera, laser, radar etc. The information from these sensors has to be processed to make the vehicle aware of the environment around. The processing involves complex mathematical algorithms that have to be implemented in the embedded software. The embedded software runs on small micro-controllers that are constrained in terms of the resources available - memory and runtime.

The development of the algorithm typically starts with the concept validation. It is usually done with powerful tools with lots of ready-to-use libraries. A good example of such tools is MATLAB and python. The concept is implemented quickly in these tools. The data from the sensors is organised and fed to the algorithm and the performance of the algorithm is evaluated. Once the concept proves to be working, it is taken to the next stage - embedded development.

The embedded development involves implementing the core concept developed above in C or in C++. The luxury of ready-to-use libraries is in most cases not available on most micro-controllers. The early stages of development happen in the Software-in-the-loop (SIL) environment. The SIL environment is a PC based environment, that is capable of reading logs of the sensor measurements stored in the PC and supply it to the algorithm. The algorithm is developed to a certain level of usability and acceptance after which it proceeds to the system integration.

The system integration of the algorithm involves directly connecting the algorithm to the data from the sensors. The system integration puts the algorithm into test with data provided on the system interface which is typically a bus from the sensor to the micro-controller that runs the code. The data is not supplied in a synchronous fashion. The processing of the algorithm might be interrupted by interrupts. The system is first put to test in a Hardware-in-the-loop set-up where a simulation tool mimics the sensor behaviour and feeds data into the system. Once the system reaches a certain degree of acceptance it is moved to vehicle test.

Overview of SIL environments

The SIL set-up that runs on the PC is a powerful mechanism to develop complex algorithms. Debug tools help the developer to step through the code and analyse a great deal of information that aids fixing issues or finding new ideas to solve problems. For algorithms such as tracking and environment perception, the SIL set-ups are tailored to provide a visual representation of the environment and the objects in them. Developers thrive in such powerful environments. The time taken to identify bugs and fix them decreases dramatically. The environment is also conducive for bringing up new ideas and quickly putting them to test their performance. However the SIL set-up suffers from certain disadvantages which are detailed below.

The SIL set-up runs on the PC and therefore cannot give an estimate of the resources required on the target micro-controller where the code is intended to run. The memory footprint required can be evaluated to a certain extent from the SIL environment. But it is not possible to evaluate the runtime required.

Some micro-controllers come with hardware accelerators that are tailor made to increase the processing speed. A good example of such an accelerator is the single instruction multiple data (SIMD). These hardware accelerators might not always provide the same results as in the PC due to subtle differences in floating point rounding or truncation operations. Some parts of the code might also be dispatched to co-processors that are faster for certain specific operations. The code that transfers the data between these units might have pitfalls which are not visible in the SIL set-up.

Overview of HIL environments

The HIL set-up [2][4] is considered to be the final step before the micro-controller is taken for vehicle testing. Typically, there is no change in software or hardware connections between the HIL and the vehicle set-up. Therefore the HIL is considered as a powerful test set-up. However, the HIL has various disadvantages during the early stages of development.

The algorithm during the development phase might not be optimised in terms of code size and stack. The algorithm may fit on the micro-controller when integrated as a standalone component on an operating system that performs just the fundamental tasks. But the HIL set-up requires full system integration with all other components and the operating system needs to support all required operations. Therefore there is a high risk that the HIL set-up cannot be used to test the performance of the algorithm in this phase of development.

In cases where the code size is under control on the target and the HIL set-up fits, the next bottleneck arises - the runtime. The code may not be optimised enough to run on the target respecting the real-time runtime requirements. Or the code is still in the evaluation phase where, it is necessary to check if the current algorithm is useful on this target. If it does not fit, then other algorithms might have to be looked into.

The HIL simulation environment typically runs on the PC or on a real-time simulator. It is not guaranteed that the system receives the same inputs from the simulation at the same time step on each iteration of the HIL [7]. Furthermore, the system might receive interrupts from other events that need not necessarily happen at

the same time on each run. A good example of such an event is the user interaction. Given these subtle variances in the HIL set-up, it is never guaranteed that the system produces identical results on every run. The most important factor for debugging and fixing an issue is the ability to reproduce it in a reliable way. As the system behaviour is not necessarily reproducible in the HIL most algorithm developers do not prefer to debug the algorithmic issues in the HIL. Nevertheless there are situations where the HIL debugging is necessary especially in situations where hardware accelerators are used for certain sections of the code to speed up the runtime.

As the runtime evaluated in the HIL set-up varies from run to run, it cannot be reliably used to track the increase or decrease in runtime for every software release. Heavy impacts on runtime due to implementation of new features can be felt. But in case there are no major features added, developers and integrators are left wondering the reason for the changes in the runtime. Figure1 shows the variations observed in the runtime of the same software with the same inputs tested in the same scenario with three runs on the HIL bench one after the other. A debugger with profiling capabilities can be used in a HIL set-up to track the code path and the time spent on each of the functions. If these functions are optimised and the code is run, then it is not guaranteed that the maximum runtime would again be hit at the same point. This gives rise to difficulties in testing the effect of optimisations on the runtime. Further, some optimisations are intended to not cause any change in the system output other than reduction of the runtime. But the HIL set-up does not provide a simple way to ensure there was no change in the system output before and after the optimisation.

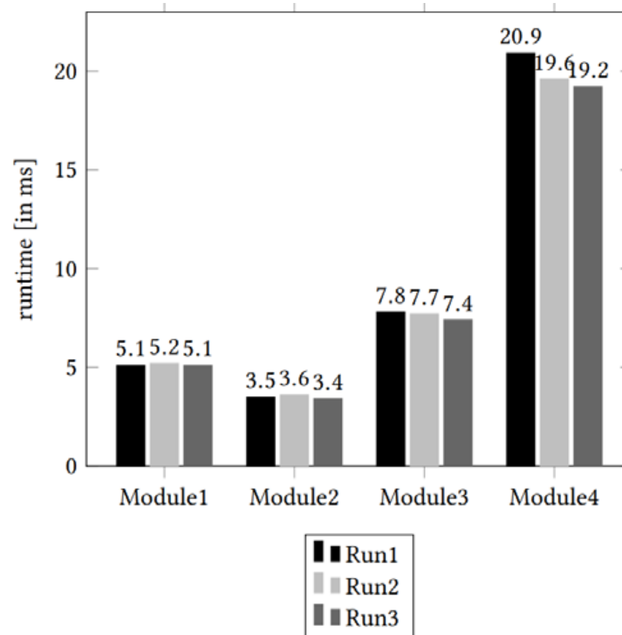

Figure 1: HIL runtime variations across runs

Recording of debug data from the HIL set-up is limited by the bandwidth available on the debug interface which is constrained for most real-time systems. It is widely thought that the HIL solves the disadvantages of the SIL. Although it is true that the HIL can reproduce the problems, it does not provide an environment conducive for debugging.

Proposed approach using PIL

To overcome the disadvantages of the HIL and SIL environments outlined in the earlier sections, an alternate method of testing is required. The intention is not to ridicule the usage of the HIL and SIL set-ups but rather highlight their deficiencies and propose additional set-ups that can help to overcome these deficiencies. Finding out and solving issues at the right place at the right time can minimize the cost of development and testing. The approach proposed here makes use of the Processor-in-the-loop (PIL) set-up, which aims to overcome the disadvantages of the HIL and add more value to the SIL. Also any new activity is always viewed upon by all stake holders - developers, integrators and managers as an additional burden. To overcome these issues the proposed PIL environment is designed and set-up in such a way that very minimal effort is required. The PIL environment is either directly or indirectly connected to the SIL and all visualisation and debug mechanisms available in the SIL are made use of. The PIL environment allows the software to be tested in its target hardware, albeit in a non real-time environment and has the following advantages:

- Earlier detection of issues during the development process:
    - Numerical instabilities (e.g. due to single-precision floating-point).
    - Compiler problems.
- The repeatability of the tests enables the possibility for runtime optimization and reliable test of bugs that are identified and fixed.
- Isolated profiling of the step that has the maximum runtime for detailed analysis.
- Less expensive than other validation methods (like HIL).
    - The test bench is inexpensive to set-up and relatively easier to debug than the HIL.

Prerequisites for the PIL

The PIL set-up is essentially a data driven environment, unlike the HIL which is time triggered. The PIL environment sleeps until all relevant data for one time step is received. Once all data for a step is available, the step is executed, the results are collected and then the system waits for the input of the next step to be received. The control for sending of the inputs and receiving the outputs is vested in the SIL environment. Therefore it is required to have a SIL set-up up and running. The advantages of the SIL set-up are explained in the SIL section and the PIL is intended to make use of all these advantages.

The next major requirement for the PIL is to have the ability to compile the code under test in the target hardware. A full system integration which is a pre-requisite for the HIL is not required. It is sufficient enough if only the code under test is compiled and is accessible.

The code under test should have well defined input interfaces, a configuration function that initialises the component, an update function that processes the inputs of every step and accumulates the component outputs along with additional debug data that is harvested from different points of the algorithm.

Another requirement is to have a means of communication between the PC which runs the SIL and the target hardware. The configuration information for initialisation and the inputs for every step of processing have to be sent from the PC to the target.

The outputs of the algorithm along with the debug information have to be transferred from the target to the PC. Therefore this communication link determines the speed at which the PIL can run. Most small micro-controllers do not have high speed communication interfaces. In such cases hardware debuggers provide interfaces through which the data can be written directly into specific memory sections.

The last requirement is to have a protocol to transfer the necessary data back and forth between the PC and the target. The PC and the micro-controller need not necessarily have the same hardware architecture. This leads to issues with endianness. Furthermore, the padding and alignment of complex data structures are not necessarily identical. This means that the data cannot simply be transferred between the two blocks just by dumping the memory. A serialization protocol for data transfer is required.

To summarize, the following blocks are required for the PIL environment.

1. A working SIL environment to supply inputs and collect outputs.
2. Code under test compiled and flashed on target hardware with a simple operating system that supports the minimum interfaces that are required for the PIL.
3. Inputs, outputs and debug data of the code under test.
4. Communication link up and running between PC (SIL) and target hardware.
5. Communication protocol for data transfer - serialisation and de-serialisation.

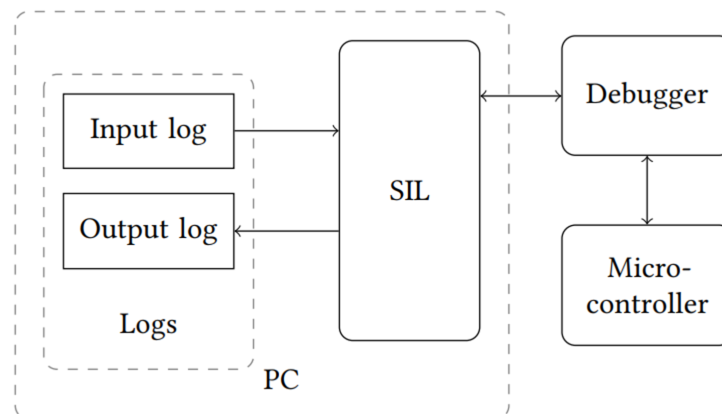

Figure 2: PIL set-up

PIL set-up

A standard PIL set-up includes the PC that has access to the logs of the input data and can run the SIL, the evaluation board with the target micro-controller and a hardware debugger as the interface between them. The evaluation board that is linked to the micro-controller with the target hardware is plugged to the power supply. A hardware debugger that can flash and communicate with the micro-controller is connected to the PC. The communication between the PC and the target hardware is done through the debugger (see Figure2). Other communication mechanisms like CAN or Ethernet can also be used. The object code is downloaded to the micro-controller via the debugger. The SIL environment is compiled to emit out the inputs required for the code under test on the target. The SIL serialises the inputs for the current time step and dispatches them to the target. The target on reception of information de-serializes the inputs and runs the code under test. The outputs and run-times are collected, serialised and sent back to the SIL. The SIL is under the

direct control of the user. It can be run constantly updating every step or can be paused to have an in-depth analysis such as comparison of the results on the SIL and the PIL for the current time step. The data flow of the proposed PIL environment is shown in figure3. Tools like MATLAB also support code verification and validation with PIL simulation of model based algorithms implemented in Simulink, as described in their web-page [6].

PIL - Protocol for data transfer

The biggest challenge in the PIL set-up is to ease the process of serialising and de-serialising the data. Handwritten code for the serialisation and de-serialisation of data are cumbersome and error prone. Also, this creates additional workload to setup the environment. Errors in the serialisation process can lead to a lot of time lost in debugging to identify the source of the error and get it fixed. To overcome these issues, it is necessary to automate the process of code generation for serialisation.

Google protocol buffers [3] is a good approach to serialise the data. However, the code generated for the protocol buffers is quite heavy and difficult to run on the micro-controllers. NanoPb [1], a light weight implementation of the Google protocol buffers can be used. Nonetheless, Google protocol buffers require the generation of the proto files which define the message format. These files could also be auto-generated based on the required data structures. Yet, the approach was avoided to reduce the code size on the embedded target and not to have a two step conversion process - to proto files and then serialisation using NanoPb. Also the protocol buffers are intended for storage and re-use of data with backward compatibility. Since this is not the scope of the PIL environment, this approach was discarded and a simple approach outlined below was taken.

The process of generating the serialisation code requires collection of the data structures that form a part of the input and output of the code under test. Each item of this collection is given a specific identity. The enlisted data structures are passed through a pre-processor which identifies dependencies for these data structures. The serialisation code is generated not only for the enlisted data structures but also for their dependencies. The pre-processor also provides an abstract syntax tree (AST) representation of the data structures. The AST representation is parsed to the leaves of the tree until a primitive data type is available. Primitive data types include signed and unsigned integers of 8, 16, 32 and 64 bits, single and double precision floats. Every primitive data type is encoded into a serialised byte array depending on its byte width. If a non-primitive data structure is encountered at the leaf, the corresponding serialisation function is called. The dependency solver ensures that the serialisation function is available for these data structures.

In most cases, the inputs or outputs have to be accumulated from different sources. It is also not guaranteed that all inputs and outputs will be available in all time steps. Therefore it is necessary to have the possibility to concatenate different serialisation streams for transmission. Therefore, for all enlisted data structures, a small header is added at the beginning of the serialised stream. The header contains the id of the data structure and its length.

The process of generating the code is embedded into the build environment. When a new data structure is added to the serialisation list or when an existing data structure

in the serialisation list has undergone a change, the serialisation code is updated. It is not necessary to maintain the backward compatibility for data structure changes since the process does not involve storage of the inputs and outputs. Any change of the inputs involves an update of the SIL environment which prepares the inputs and the code under test makes use of the updated inputs. This makes the process of serialisation unaffected by any changes.

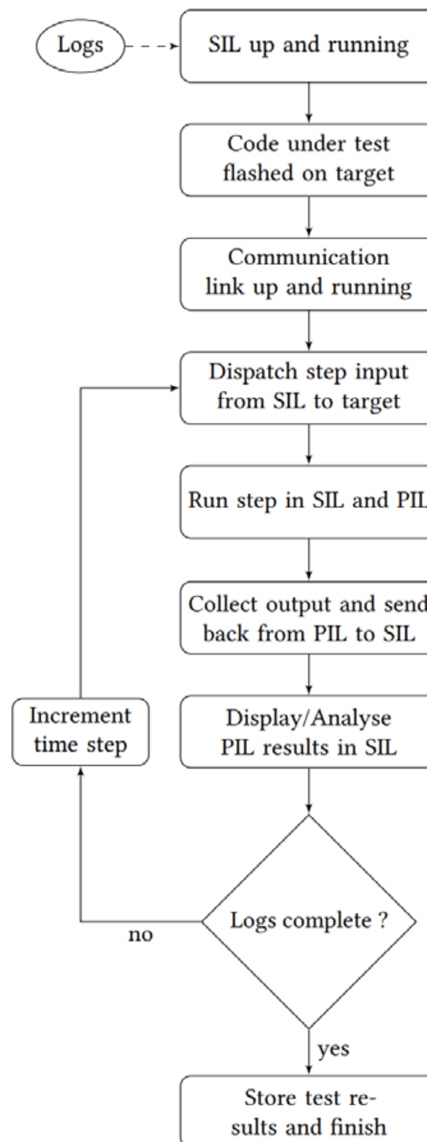

Figure 3: Data flow in a PIL set-up

Application and results

Once the PIL environment is set-up, it requires very little or no maintenance. It has different use cases to supplement the SIL environment and is a pre-cursor for the HIL environment. Some usages of the PIL are detailed in this section.

Testing and validation

The PIL set-up can be utilized to perform sanity checks if the code would produce similar results as expected in the SIL. Both the SIL and the PIL are data driven and there is no discrepancy in the inputs. This means that for fixed integer arithmetic the

results are identical. For floating point arithmetic the results may differ slightly due to the subtle differences in rounding and optimisation of intermediary operations on the PC and on the target.

The PIL set-up is also guaranteed to produce identical results on every run. Any deviation of the results can only arise due to a change in the code. If the code is identical but results differ between runs, then this situation is a good indicator that there is a flaw in the code. In most cases this is caused due to uninitialized memory access. There are also rare cases which are due to compiler bugs. During such situations where the results are not identical between runs, the debug outputs are harvested at different check points inside the code under test. This helps narrow down the location of the anomaly.

Runtime optimization

Given that executing the PIL with the same code and same inputs yields essentially the same runtime results, unlike running the software in the HIL, it can be used for optimization purposes. Typically, embedded systems have a cyclic update rate. Since the PIL runs step by step, the time step with the maximum runtime can be identified by just looking at the runtime of each step. Once the step is identified, it is important to know what exactly is the code path taken during this step. This can be done by analysing the debug data recovered in this time step. An alternative way is to make use of the profiler that might be available with the debugger for the target. The profiler would provide information about the exact code path taken, the number of times a particular function has been executed and the runtime of each of those runs. Profiling is also possible in HIL set-ups. But the fact that HIL runtime cannot be reliably reproduced makes the PIL all the more powerful for runtime optimisation. The code path taken in this step is analysed and possible optimisations are implemented. Then the PIL is rerun. If the optimisation is not intended to have any changes to the results, then this can be easily evaluated by comparing the results of the runs before and after the optimisation. Once successful, the next step with the highest runtime is dealt with until the desired level of performance is achieved.

Regression testing

The software engineering processes in most automotive industries employ agile mode of development. This means releases happen far more frequently than traditional embedded software. It is important to keep the runtime of the different systems under required levels for a smooth performance of the system. The PIL is an ideal candidate for these checks. The input from the SIL for each time step is written to a file. Before every release, the PIL is automatically run with the pre-defined input file and the comparison of the runtime is done with the previous release. An acceptance criterion is set for allowed increase in runtime. The developer is flagged immediately when the runtime exceeds the set threshold. Investigation can begin immediately to ascertain the cause for the increase in runtime. In some cases the increase in runtime might be legitimate due to the implementation of new features. However, it has to be ensured that the runtime is within the limits of the system and the PIL provides important inputs in achieving it.

Comparison and/or evaluation of different targets

The PIL is also a useful mechanism for comparing or benchmarking different micro-controllers. Gone are the days when the clock frequency of the micro-controller

could be used to estimate the processing power. For example, faster memory access can speed up the runtime to a great extent. Single instruction, multiple data (SIMD), ARM Neon architectures on ARM devices etc. can reduce the processing time required by a huge factor. The neon intrinsic on the ARM allows the computation of four floating point operations simultaneously. This means that matrix operations, especially matrix multiplication, can be executed nearly four times faster than it would be done without the neon intrinsic.

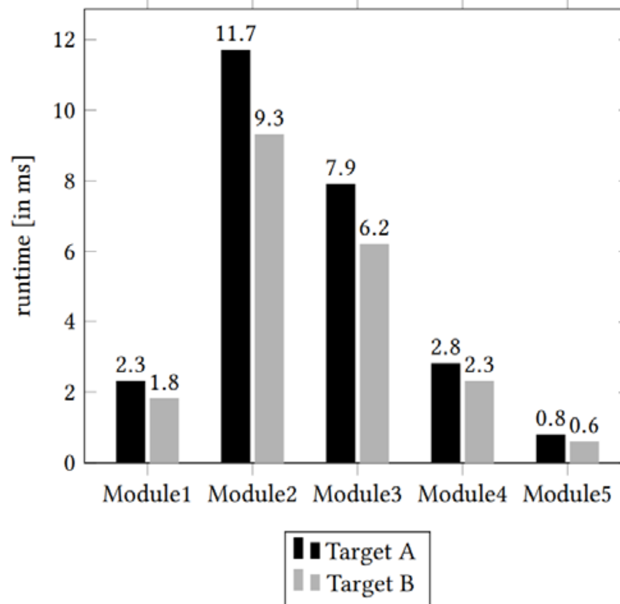

Figure 4: Runtime Comparison

The evaluation of runtime on the target can help new projects choose the hardware that would fit the purpose. It also helps to evaluate if the algorithm can run on the target hardware or a different algorithm has to be implemented in order to fit on to the target hardware. The comparison of the runtime of a system on two different micro-controllers is shown in figure 4. Both targets are different micro-controllers running at a clock speed of 120 MHz. The PIL test results show that target B is about 25% faster than target A. This is an important indication that if a project decides to switch from target A to target B, there is a decrease in runtime which can be made available for other functionalities.

Conclusions

The paper outlines the deficiencies of the SIL and HIL environments in the scope of implementation and validation of complex mathematical algorithms in micro-controllers. An additional approach is proposed to make use of the PIL environment in combination with the SIL. Mechanisms are implemented to minimise the effort required to set-up such PIL environment. Furthermore, the generation of the data format for communication between the PC and the target is embedded into the software build process. The risk of error in data conversion and transfer is thereby eliminated. The PIL environment provides early insights to the ability of the hardware to run the algorithm in the available runtime. Optimisations can be implemented and tested directly on the target hardware. It also helps in comparison and benchmarking of different micro-controllers.

References

[1] Petteri Aimonen. NanoPb. https://jpa.kapsi.fi/nanopb/

[2] M. Bacic. 2005. On hardware-in-the-loop simulation. In Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC ’05. 44th IEEE Conference on Decision and Control. IEEE, Seville, Spain, Spain. https://doi.org/10.1109/CDC.2005.1582653

[3] Google Protocol Buffers. http://code.google.com/apis/ protocolbuffers/

[4] R. Boot ; J. Richert ; H. Schutte ; A. Rukgauer. 1999. Automated test of ECUs in a hardware-in-the-loop simulation environment. In Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium. IEEE, Kohala Coast, HI, USA. https://doi.org/10.1109/CACSD.1999.808713

[5] Luiz S. Martins-Filho, Adrielle C. Santana, Ricardo O. Duarte, and Gilberto Arantes Junior. 2014. Processor-in-the-Loop Simulations Applied to the Design and Evaluation of a Satellite Attitude Control. In Computational and Numerical Simulations, Prof. Jan Awrejcewicz (Ed.). InTech. https://doi.org/10.5772/57219

[6] MathWorks [n. d.]. Code Verification and Validation with PIL and External Mode. Retrieved April 30, 2018 from https://de.mathworks.com/help/supportpkg/beaglebone/examples/ code-verification-and-validation-with-pil-and-external-mode.html

[7] H. Thane ; D. Sundmark ; J. Huselius ; A. Pettersson. 2003. Automated test of ECUs in a hardware-in-the-loop simulation environment. In Proceedings International Parallel and Distributed Processing Symposium. IEEE, Nice, France. https://doi.org/ 10.1109/IPDPS.2003.1213515

Author

Anto Michael is a software architect at Valeo specialising in the implementation of complex mathematical algorithms in embedded software on microcontrollers for driver assistance systems in passenger cars. He is also a system expert in the automated parking and low speed manoeuvring product segment. He has about 10 years of relevant industry experience.

Contact

Email: anto.michael@valeo.com