Designing a Mathematically Verified I^2C Device Driver Using ASD

Arjen Klomp [a 1], Herman Roebbers [b], Ruud Derwig [c] and Leon Bouwmeester [a]
[a] *Verum B.V., Laan v Diepenvoorde 32, 5582 LA, Waalre, The Netherlands*
[b] *TASS B.V., P.O. Box 80060, 5600KA, Eindhoven, The Netherlands*
[c] *NXP, High Tech Campus 46, 5656 AE, Eindhoven, The Netherlands*

Abstract. This paper describes the application of the Analytical Software Design methodology to the development of a mathematically verified I^2C device driver for Linux. A model of an I^2C controller from NXP is created, against which the driver component is modelled. From within the ASD tool the composition is checked for deadlock, livelock and other concurrency issues by generating CSP from the models and checking these models with the CSP model checker FDR. Subsequently C code is automatically generated which, when linked with a suitable Linux kernel run-time, provides a complete defect-free Linux device driver. The performance and footprint are comparable to handwritten code.

Keywords. ASD, CSP, I^2C, Device Driver, Formal Methods, Linux kernel

Introduction

In this Analytical Software Design [1] [2] (ASD) project, NXP's Intellectual Property and Architecture group successfully used Verum's ASD:Suite to model the behaviour of an I^2C [3] device driver. It had been demonstrated in other case studies [4][5] to save time and cost for software teams developing and maintaining large and complex systems. But this was the first time the tool had been applied to driver software.

NXP is a leading semiconductor company founded by Philips and, working with Verum, undertook a project to evaluate the benefits of using ASD:Suite for upgrading its device driver software.

1. Background

Analytical Software Design (ASD) combines the practical application of software engineering mathematics and modelling with specification methods that avoid difficult mathematical notations and remain understandable to all project stakeholders. In addition, it uses advanced code generation techniques. From a single set of design specifications, the necessary mathematical models and program code are generated automatically.

ASD uses the Sequence Based Specification Method [6] to specify functional requirements and designs as black box functions. These specifications are traceable to the original (informal) requirements specifications and remain completely accessible to the critical project stakeholders. In turn, this allows the stakeholders and domain experts to play a key role and fully participate in verifying ASD specifications; this is an essential requirement for successfully applying such techniques in practice.

[1] Corresponding Author: *Arjen.Klomp@Verum.com*

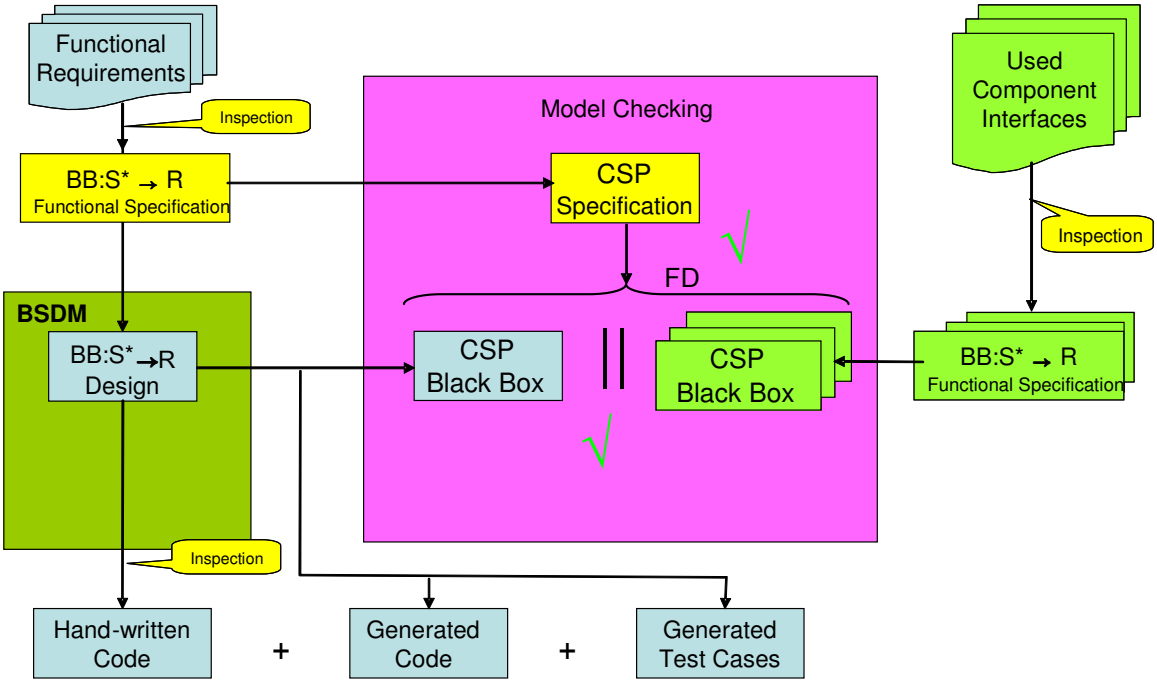

Figure 1. An overview of Analytical Software Design.

At the same time, these specifications provide the degree of rigour and precision necessary for formal verification. Within ASD, one can apply the Box Structured Development Method (BSDM) [7], following the principles of stepwise refinement, to transform the black box design specifications into state box specifications from which programming is typically derived.

Figure 1 summarizes the main elements of ASD. The functional specification is analysed using the Sequence Based Specification method extended to enable non-determinism to be captured. This enables the externally visible behaviour of the system to be specified with precision and guarantees completeness.

Next, the design is specified using Sequence-Based Specification. This still remains a creative, inventive design activity requiring skill and experience combined with domain knowledge.

With ASD, however, the design is typically captured with much more precision than is usual with conventional development methods, raising many issues like illegal behaviour, deadlocks, race conditions, etc. early in the life cycle and resolving them before implementation has started.

The ASD code generator automatically generates mathematical models from the black box and state box specifications and designs. These models are currently generated in CSP [8], which is a process algebra for describing concurrent systems and formally reasoning about their behaviour. This enables the analysis to be done automatically using the ASD model checker (based on FDR [9]). For example, we can use the model checker to verify whether a design satisfies its functional requirements and whether the state box specification (used as a programming specification) is behaviourally equivalent to the corresponding black box design.

In most cases, a design cannot be verified in isolation; it depends on its execution environment and the components it uses for its complete behaviour. In ASD, used component interfaces are captured as under-specified sequence based specifications. The corresponding CSP models are automatically generated, with under-specified behaviour

being modelled by the introduction of non-determinism. These models are then combined with those of the design and the complete system is verified for compliance with the specification. Defects detected during the verification phase are corrected in the design specification, leading to the generation of new CSP models and the verification cycle being repeated (this is typically a very rapid cycle).

After the design has been verified, the ASD code generator can be used to generate program source code in C++, C or other similar languages.

Section 2 introduces the case study and section 3 is about the method used. In Section 4, we give a more detailed overview of how ASD techniques were applied in practice for this particular case and section 5 presents the resulting analysis performed.

2. Case-study: a Mathematically Verified I^2C Driver

2.1 Introduction

NXP's IP and Architecture group was developing code for an I^2C device driver, for next generation systems. I^2C drivers had been in the marketplace for a number of years. In theory, the existing software can be reused for new generations of the hardware, but in practice there were timing and concurrency issues. Therefore they also wanted to know whether the ease of maintenance of the driver could be improved.

To learn more about the capabilities of the ASD:Suite and its underlying technology, the group carried out a joint project with Verum in which ASD was applied to model the behaviour of the device driver.

The main objectives of the case study were to:

- Verify the benefits of ASD. The most important benefits of interest to NXP were:
    - Delivering a higher quality product with the same or reduced amount of development effort.
    - Reducing costs of future maintenance.
    - Achieving at least equivalent performance to the existing product.
- Determine if ASD modelling helps in verifying the correctness of device driver software.
- Assess whether the ASD:Suite is a useful and practical tool for developing device drivers.

This paper presents an overview of how the ASD method and tool suite were applied to the development of a Linux version of the IP_3204 I^2C driver software, hosted on NXP's Energizer II chip, and draws conclusions based upon these objectives.

2.2 Current Situation

I^2C devices are used particularly in larger systems on chips (SoCs). These are typically employed in products such as digital TVs, set-top boxes, automotive radios and multimedia equipment, as well as in more general purpose controllers for vending machines etc. Quality is a major concern, especially for the automotive market, where end user products are installed in high-end, expensive vehicles.

The I^2C driver software had been ported and updated many times, both to keep pace with the evolving hardware platform and to cater for the requirements of new and upgraded target operating systems. The existing device driver in the past suffered from timing and concurrency issues that caused problems in development and testing, largely stemming from an incomplete definition of the hardware and software interfaces.

3. Method

ASD:Suite had been demonstrated in other projects to shorten timescales and to reduce costs when used in the development and maintenance of large and complex systems. This was, however, the first time it had been applied to device driver software.

The project covered three key activities:

- Modelling of the device driver software.
- Automatic generation of C code.
- Integration into the Linux Kernel.

NXP provided the domain knowledge for the project. Verum did the actual modelling of the driver specification and design. TASS provided input for the C code generation process and assisted in the implementation of the Linux kernel Operating System Abstraction Layer (OSAL) including interrupt management.

3.1 Modelling the Hardware and Software Interfaces

The dynamic behaviour of the original driver was largely unspecified and unclear in the original design. The application of ASD clarified the interfaces, which resulted in a better understanding of the behaviour. For example, the software interface, which NXP calls the hardware API (HWAPI), was assumed to be stateless. However, ASD modelling revealed HWAPI state behaviour that had not previously been documented.

3.2 Improving the Design

The design of the original I^2C driver software was different. More work was done in the interrupt service routine context, instead of in the kernel threads. Using ASD it was possible to do the majority of the driver work in a kernel thread and as little as possible in the interrupt service routine. There is trade-off here; doing more in the kernel thread will improve overall system response but can reduce the driver performance itself. ASD will however ensure that the behaviour of the driver is complete and correct.

3.3 ASD Methodology

At all stages of a project, ASD technology is applied to the various parts of the device driver software, and can also be applied to the hardware. In summary the approach, visualized in Figure 2 is:

1. Using the ASD:Suite, gather and document requirements, capturing them in the ASD model.

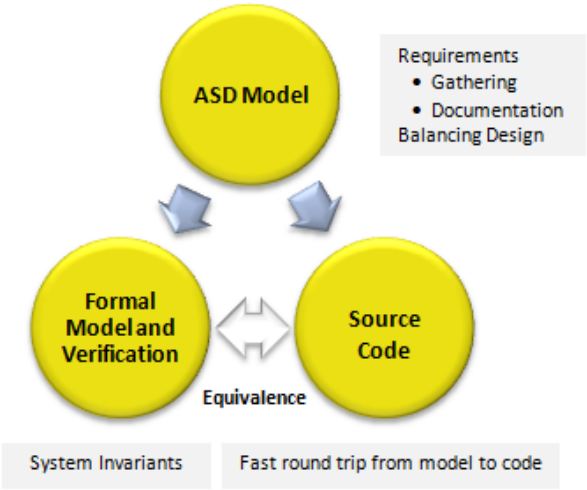

Figure 2. The triangular relation between ASD Model, CSP and source code.

2. Balance the design for optimal performance (e.g. by introducing asynchronous operation).

3. Generate a formal model of system behaviour and verify the correctness of the design (before any code has been generated).

4. Generate the source code, guaranteed to be defect free and functionally equivalent to the formal model.

3.4 Key Benefits

When a software developer has gained familiarity with the ASD:Suite, it is possible to improve the efficiency of the development process by reducing dependency on testing, cutting development timescales by typically 30%. Maintenance becomes easier because the code generated is operating system independent, and changes are implemented to the model, not the code, which can then be easily regenerated.

ASD:Suite delivers high performance software with a small code footprint and low resource requirements which is suited to highly embedded systems. Most importantly, the quality of the software improves, because code defects are eliminated.

4. Integrating ASD Generated Code with the Linux Kernel

In general the structure of an ASD application can be described as depicted in Figure 3 below. The ASD clients call ASD components, which communicate with their environment through calls to the ASD Run Time Environment (RTE). This RTE uses an OS Abstraction Layer (OSAL) to make calls to some underlying OS. This should provide easy porting from one OS to another. The OSAL comprises a set of calls to implement thread based scheduling and condition monitors, as well as time related functions used by the RTE. In case of a normal application program the OSAL is directly mapped onto POSIX threading primitives, mutexes and condition variables in order to provide the required scheduling and resource protection. Currently only the C generator uses the OSAL interface.

In the case of the I^2C driver for Linux there is more to do than just have a standard POSIX implementation for the ASD Run Time Environment, as this now has to operate in

kernel space and implement a device driver. What this means is that we need to have a standard driver interface (i.e. open, read, write, ioctl, close) on top of the ASD components and that the ASD client needs to use the corresponding system calls to communicate with the ASD components from user space.

Furthermore we need to talk to the hardware directly, which means that we need to deal with interrupts in a standard way. This is realized by an ASD foreign component. A foreign component is the ASD term for a component for which the implementation is handwritten code that is derived from the ASD interface model.

4.1 Execution Semantics of ASD Components

In order to understand the requirements of the RTE one needs to consider the execution semantics of ASD components.

4.1.1 ASD Terminology

We introduce the concept of durative and non-durative actions. A durative action takes place when a client synchronously invokes a method at a server. Until this method invocation returns the client remains blocked and it cannot invoke other methods at the same or other servers. After the method has returned to the client, the server *is going to process* the durative part and eventually the server informs the client asynchronously through a callback. A non-durative action takes place when a client synchronously invokes a method at a server and remains blocked until the server *has completely processed* the request. Basically an ASD component can offer two different kinds of interfaces: synchronous ones and asynchronous ones: The synchronous interfaces are client interfaces and used interfaces and the asynchronous interfaces are the callback interfaces. A component can offer several client interfaces, use several server interfaces and offer as well as use several callback interfaces. In order to be able to deal with asynchronous interfaces there is a separate Deferred Procedure Call (DPC) thread per callback interface. Making a call to a callback interface is implemented as posting a callback packet in a DPC queue serviced by a DPC thread, and notifying this DPC thread that there is work to do. The component code executes in the thread of the caller (it is just a method/subroutine call).

Having the DPC threads execute as soon as they are notified may create a problem when there is shared data between the component thread and the DPC thread. As there always is shared data (state machine state), this data needs to be protected from concurrent access using some form of resource protection.

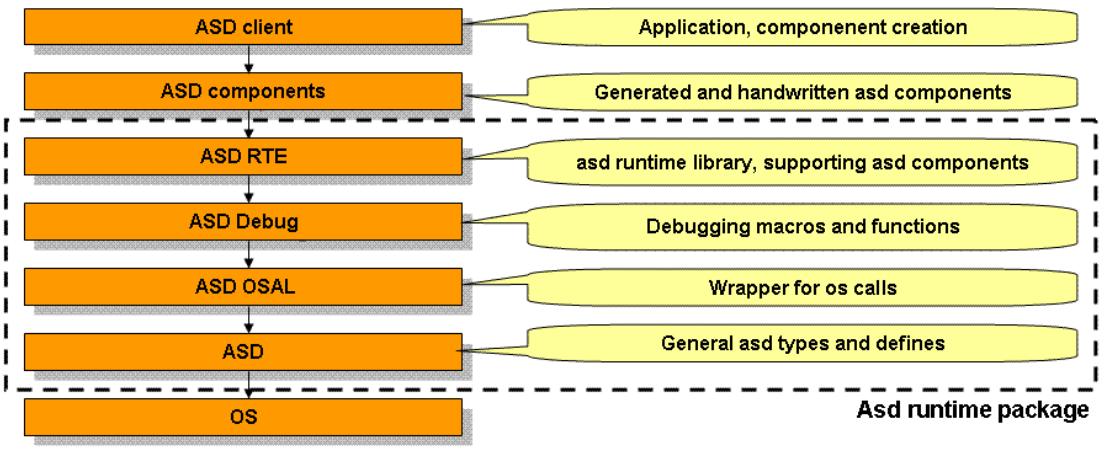

Figure 3. General overview of an ASD system.

For ASD this amounts to run-to-completion semantics. What this means is that it is guaranteed that the stimulus and all responses have been processed completely in the specified order and all predicates are updated before the state transition is made.

This then implies that only one client may be granted access to the component at any time, as well as that DPC access can only occur after completion of a synchronous client call. In other words: an ASD component has monitor semantics.

Furthermore DPC threads must execute and empty their callback queues before new client access to the component is granted. This is realized by the Client-DPC synchronization.

Finally, special precautions must be taken for the following scenario: A client invokes a non-durative action on the interface of component A, where run-to-completion must be observed. If this invocation results in the invocation of a durative method and the client can only return when a callback for the durative action is invoked, we would get into trouble if the DPC server thread would block on the synchronization between client and server thread. In order to prevent this scenario there is a conditional wait after the release of the client-DPC mutex. When the client leaves the component state machine it needs to pass the conditional wait, to be released by the DPC server thread after this has finished processing the expected callback.

The following pseudo code enforces the client side sequence of events with the minimum amount of thread switching and is depicted in Figure 4:

```
Get ClientMutex
  Get ClientDPCMutex
        CallProcessing
  Release ClientDPCMutex
  ConditionalWait (DPC callback)
Release ClientMutex
```

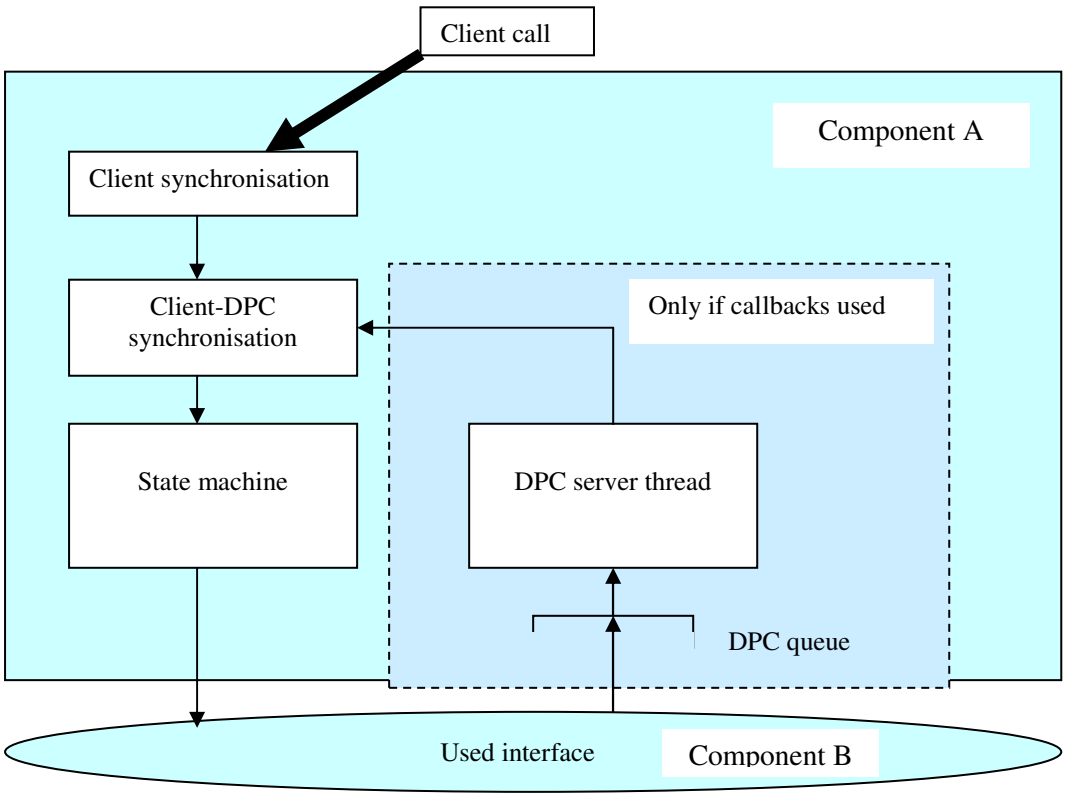

Figure 4. Internal structure of an ASD component.

The DPC thread, shown inside the dotted area, executes the following pseudo code:

```
while (true)
{
  WaitEvent(wakeup, timeout)
  Get ClientDPCMutex
        CallProcessing
  Release ClientDPCMutex
  Signal DPC callback
}
```

4.2 Structure of an ASD Linux Device Driver.

When using an ASD component as a Linux device driver some glue code is necessary. The standard driver interface must be offered to the Linux kernel in order that the driver functions may be called from elsewhere. Also the implementation of the ASD execution semantics is not so straightforward, because we now need to consider hardware interrupts from the device we are controlling. As ASD cannot directly deal with interrupts we need a way to convert hardware interrupts to events that ASD can cope with. In effect we need an OS Abstraction Layer implementation for Linux kernel space. This means that instead of user level threads we now need kernel level threads, and we need to decouple interrupts from the rest of the processing. To this end we use a kernel FIFO, in which either the interrupt service routine or an event handler writes requests for the kernel level event handler. The kernel FIFO takes care that posting messages to and retrieving messages from the FIFO is interrupt safe. The first level event handler is implemented as a kernel work queue function scheduled by the interrupt handler, which reads messages from the kernel FIFO and then puts messages in a message queue serviced by a kernel DPC thread. This first level event handler is not allowed to block. The kernel DPC thread, signalled by the first level event handler, *is* allowed to block. Figure 5 explains this in more detail.

Interrupt

```
i2c_isr() {
   event = deal_with_IRQ();
   kfifo_put(event);
   queue_work(
      interrupt_wq,
      &i2c_event_work)
}
```

workqueue function

```
i2c_event_handler(){
  while (msg_in_kfifo()){
     kfifo_read(…,&intdata,…);
     /* put msg in DPC queue
      and schedule DPC thread */
     schedule_DPC_callback(&int_data);
  }
}
```

Figure 5. Connecting interrupts to ASD callbacks.

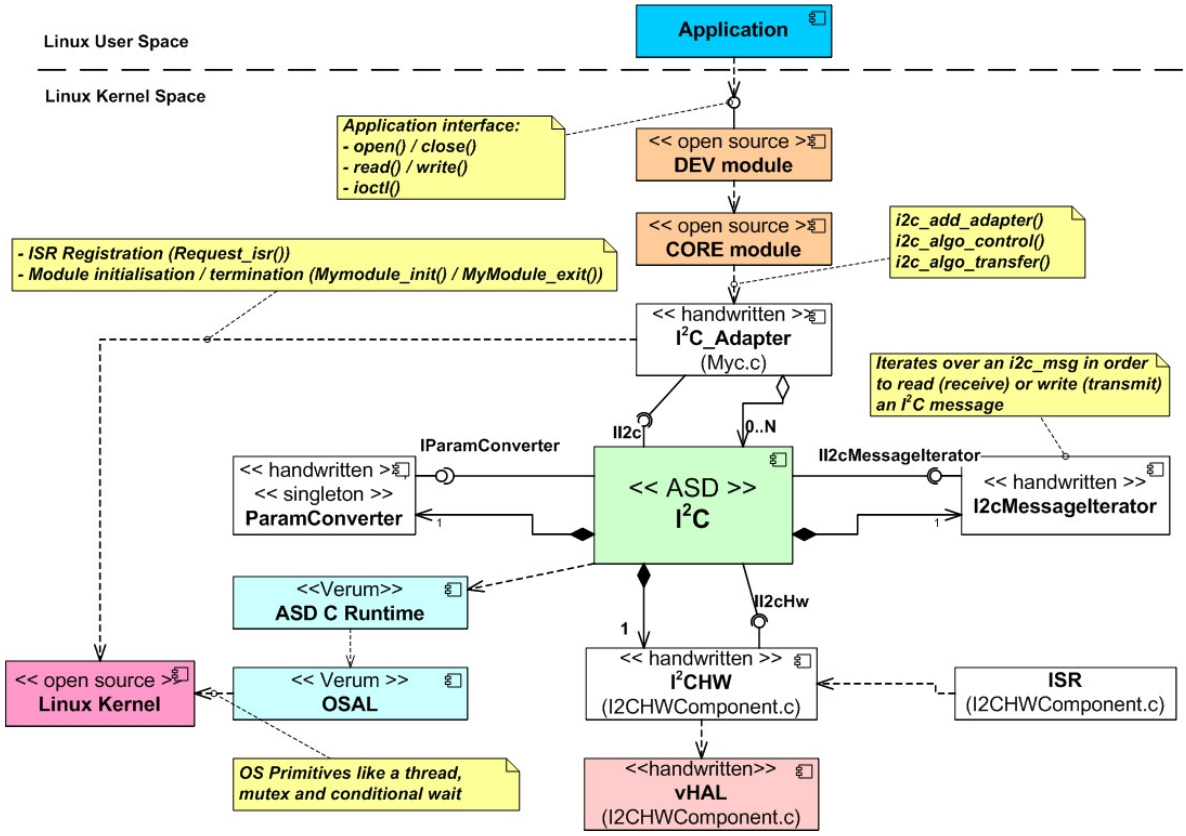

Figure 6. Structure of the ASD device driver.

Figure 6 depicts the relation between kernel code, OSAL, ASD generated code and handwritten code.

Starting from the upper layer downwards we find a standard implementation of a device structure initialisation in the dev module, offering the `open`, `close`, `read`, `write` and `ioctl` calls.

The I2C core module is also standard, calling the functions `i2c_add_adapter`, `i2c_algo_control` and `i2c_algo_transfer` implemented in adapter file `myc.c`. This file makes the connection between driver entry points and ASD generated I2C component code. In order that the component can be configured (e.g. set the correct I2C bus speed) there needs to be a way to hand configuration parameters down to a configuration function in the ASD file. `ParamConverter` converts between speed in bits/sec and the register values necessary to obtain the desired speed. `I2cMessageIterator` repeats the actions necessary for reading or writing of a single byte until the requested number of bytes is reached.

For implementing the ASD execution semantics, the ASD C RunTime Environment is used. To fulfill its job, this RTE calls upon an OSAL implementation for Linux kernel space, which this project is the first to use. The OSAL uses the Linux kernel to implement the required scheduling, synchronisation and timing functionality. The `I2CHwComponent` implements the interface to the I2C hardware, supported by a Hardware Abstraction Layer (vHAL) implementation already available for this component. This hardware component implementation is based on an interface model derived from the IP hardware software interface documentation. This interface model of the hardware is used to verify the driver. The difficult bit is, of course, to make the model behave exactly the same as the hardware.

5. Results

NXP has measured whether two of its key objectives had been met. Following completion of the project, they carried out:

1. Extensive stress testing, to check the stability and robustness of the product, and that the functionality was correct.

2. Performance analysis, in terms of speed of operation and footprint.

The results of these investigations are as follows. The original code and data sizes of the handwritten code can be seen in Table 1.

Table 1. Code and data size of original NXP driver code.

Original code	Text	Data	Bss	Total
built-in.o	12468	16	592	13076

This handwritten code is directly linked into the kernel. In order to facilitate testing the ASD driver was implemented as a loadable kernel module. Using C code generated from ASD models using the C-code generator (beta version) and combining this with handwritten code to interface to the I^2C hardware we get the results shown in Table 2.

Table 2. Code and data sizes for ASD generated C code + driver + ASD OSAL.

Type of Code	text	data	bss	total
Handwritten ASD runtime lib incl. OSAL	4824	0	0	4824
Handwritten code	4876	612	864	6352
ASD generated code	12048	40	0	12088
				+
Total code in mymodule.o	21748	652	864	23264
Final kernel module for Linux file manager				
mymodule.ko	21840	920	864	23624

We can see from these tables that the difference between the code sizes is about 10 Kbytes. This difference is constant, since the implementation of the OSAL and ASD RTE is not dependent on the driver. From inspection of the handwritten I2CHwComponent it is to be expected that there is room for optimization, which could make its code size significantly smaller. Code size optimization was, however, not the goal of this project. It will be considered for further work. Initial benchmarking has also shown that the performance of the code is acceptable, only minimally slower than the handwritten code. This is depicted in Table 3 below.

Table 3. Comparison of execution times.

Execution time	Handwritten old driver	ASD Generated code + OSAL
Send of 2 bytes	380 microseconds	386 microseconds
Time in interrupt	60 microseconds	20 microseconds

Several remarks can be made about these results. One: ASD components in general provide more functionality. They also capture “bad weather” behaviour that is not always captured or correctly captured in conventional designs. This results in many cases in a small increase in code size. Two: The way the C code is generated from the model is very straightforward. Work is underway to optimize the generated code so that it will be smaller. Even now, the larger code size is acceptable for NXP. Three: The time spent in interrupt context is more in the existing handwritten case because more is done in interrupt context than in the ASD case, resulting is a slightly faster performance. The ASD code blocks interrupts for a shorter time than the existing handwritten device driver code does, resulting in better overall system response.

Despite these positive findings, there are still some concerns:

- The ASD driver currently implements less functionality than the original driver (no multi-master, no chaining). Adding this functionality will have more impact on code size.

- Initially it did not survive a 3 hour stress test.

During integration and test of the ASD driver, a number of flaws where discovered. Some were related to mistakes in the handwritten code and some to modelling errors due to misinterpretation of the hardware software interface. The remaining stress test stability issue was determined to be caused by unexpected responses from the I^2C EEPROM used during the stress test. The driver did not expect the response from the EEPROM when writing to a bad block. With a fresh EEPROM there are no problems. Thus, the model needs to be enhanced to cope with this situation, which should be a comparatively simple exercise.

6. Conclusions

NXP believes ASD:Suite can also provide major benefits for developing defect free device drivers. The structured way of capturing complete requirements and design enable its software developers to model system behaviour correctly, verify the model and automatically generate defect free code. They are already modelling hardware, and are looking at opportunities to combine this with ASD software models, since the hardware-software interface and (lack of) specification of the corresponding protocols remains a source of errors.

This project has clearly shown that ASD modelling helps in developing and verifying deeply embedded software and that using the C code generator is beneficial and practical for device drivers.

The biggest advantage seen is the rigorous specification process that is enforced with ASD. Software designers are forced to think before they implement, and ASD helps them ensure a complete and correct specification.

It was not possible in this particular case to make a direct comparison of development effort with and without ASD, but other studies have shown that using ASD:Suite can reduce development time and cost by around 30%. Additional benefits include much easier and less costly maintenance.

NXP’s own investigations have demonstrated the quality of the product and the performance of the generated C code.

Even where there are requirements with timing constraints that cannot be modelled using ASD, race conditions and deadlocks due to unexpected interleaving of activities are prevented by the model checker, and it is a major advantage for developers to be able to perform manual timing checks on guaranteed defect free code.

Because the code generator produces verifiably correct code, the number of test cases developed and needed to run to gain confidence in the final product was considerably less than it would have been using a conventional approach to software development. The model checker revealed that there were more than 700,000 unique execution scenarios for the device driver. Without ASD, it would have required over 700,000 test cases to thoroughly test the software. Thus a major reduction in testing effort was achieved.

7. Future Work

Some thoughts have been expressed as to whether the current OSAL interface models the ASD principles in the best way. Viewing an ASD component as a separate process, and using channels to implement interfaces could be a more appropriate model for more modern OSes (QNX, OSEK, (µ-)velOSity, Integrity®), which offer higher level message passing primitives. It would also make the system scalable and offer a more efficient implementation under this kind of OS. For OSes not offering the message passing primitives mutexes and conditions can then be used to implement the required functionality. This thinking is, however, still conceptual, and not in the context of this NXP project.

References

[1] Philippa J. Hopcroft, Guy H. Broadfoot, Combining the Box Structure Development Method and CSP, In *ASE '04: Proceedings of the 19th IEEE international conference on Automated Software Engineering, pages 340–345*, Washington, DC, USA, 2004.

[2] An introduction to ASD, http://www.verum.com/resources/papers.html

[3] NXP, I^2C-bus specification and user manual Rev 3, `www.nxp.com/acrobat_download/usermanuals/UM10204_3.pdf`, NXP, 2007.

[4] Rutger van Beusekom, Nanda Technologies IM1000 Wafer Inspection System, Verum White Paper Study, *Verum*, 2008

[5] R. Wiericx and L. Bouwmeester, Nucletron uses ASD to reduce development and testing times for Cone Beam CT Scan software, Verum White Paper Study, *Verum*, 2009

[6] S.J. Prowell and J.H. Poore, Foundations of Sequence-Based Software Specification. *IEEE Transactions of Software Engineering, 29(5):417-429*, 2003

[7] H.D. Mills and R.C. Linger and A.R. Hevner. Principles of Information Systems Analysis and Design. *Academic Press*, 1986

[8] C.A.R. Hoare. Communicating Sequential Processes. *Prentice Hall*, 1985

[9] Formal Systems (Europe) Ltd, Failures-Divergence Refinement: FDR2 User Manual, 2003