Formal Specification of an Auctioning System Using VDM++ and UML, an Industrial Usage Report

Manuel van den Berg, Marcel Verhoef, Mark Wigmans
{Manuel.vanden.Berg, Marcel.Verhoef, Mark.Wigmans}@chess.nl

Chess Information Technology, P.O. Box 5021, 2000 CA Haarlem, The Netherlands

Abstract. Selecting the proper method and tools for designing a real–time embedded and distributed application is a difficult job. Evaluation of practical experiences is probably the most important means to overcome this problem. This paper presents our experiences with the application of VDM++, (real–time) UML and design patterns in an industrial environment. The combination of these formal and informal techniques has been used to design and implement the core of a new generation auctioning system for one of our clients. First, the project context will be presented. The experiences gained are then evaluated against three well–known publications on the use of formal methods, "Ten Commandments of Formal Methods" [4], "Seven Myths of Formal Methods" [9] and "Seven More Myths of Formal Methods" [3]. Finally, we present our lessons learned and draw our conclusions from the evaluation.

Project context

Chess is a Dutch–based group of companies active as a full life–cycle IT provider in the area of embedded, technical & multi–media information, communications and computer technology. We provide knowledge and expertise for design, development, integration, production and maintenance of high–end integrated circuits, computer hardware, software and integrated systems. In this particular case, we are hired to assist in the re–engineering of an existing auctioning system.

This auctioning system is the primary business process of our client. Each auction day, 50,000 transactions are made on average, generating approximately 4 million EUR daily turnover. The core of the auction system is the so–called clock system. This system

– controls an auction clock in the auction theatre,
– it interacts with all potential buyers,
– it distributes real–time data to all parties involved and
– it generates transactions to the financial and logistical systems in the back–office.

The entire auction operates 13 clocks divided over 3 auction theatres.

The clock system implements the so–called "Dutch auction" process. The auction master, the operator of the auction process, puts the clock arrow at a certain starting position. This indicates the maximum price for bidding. When the clock arrow is released, it rapidly moves down (counter clockwise) to indicate a lower price at each position, e.g. 5 cents per clock–tick. The clock moves downwards at a rate of 30 clock positions per second. The first buyer in the auction theatre that presses the "buy" button on his terminal, stops the clock and buys the product that is for sale for the price that is indicated by the arrow on the clock. By means of a voice connection the auction master and the buyer agree on the amount of the product that is bought. A complete cycle

– setting the clock in the starting position,
– releasing the clock arrow,
– detecting a buyer,
– confirming the sale by voice negotiation and

– setting up the clock for the next cycle by putting the clock arrow back up in the starting position

takes on average three seconds. The clock system is in charge of synchronising and controlling all attached subsystems.

Specifying the auction clock system

The reason for the re–engineering effort of the existing clock system is the fact that the auction wants to provide their buyers with the opportunity to participate in the auction process from any location in the world, using a personal computer connected to an ISDN telephone line. A special extranet and an auction application have been developed for this, enabling the buyers to participate in the auction in real–time (including the audio connection between auction master and buyer). The system that interacts with the existing clock system and a new extranet is the subject of this paper.

During the design of this system, we had to solve the following design challenges:

– to get a clear and concise specification of the temporal properties of the system, for instance the detection of buyers;
– to get a clear and concise specification of the functional properties of the system, for instance the processing and distribution of the real–time data flows;
– to analyze the impact of the chosen system hardware configuration (multiprocessor VME system), network and commercial–off–the–shelf real–time operating system (VxWorks) on the software architecture whilst not relying on this specific configuration.

Due to the mission critical character of the application, a lot of effort has been put into the selection of tools and techniques to tackle these design challenges. The aim of the project was to create a unified (and extensible) view on the above mentioned requirements, so that it can be used as a baseline specification for the implementation of the new clock subsystem by a team of system programmers that are no experts in formal methods.

From the start of the project it was clear that we needed a mix of formal and informal methods. The project was already well under way when the team was formed to start the software design of this subsystem. There was no time to get everyone at the proper level of formal methods knowledge but the client was constantly reminding us of the mission–critical nature of the application. Finding a good balance between formal and informal techniques, as well as choosing the right abstraction level in the formal specification has therefore been a major challenge.

Specifying a real–time application normally puts an emphasis on the temporal properties (external behaviour) of the system, while distributed applications typically highlight the functional properties, such as data transformations. Embedded applications are mainly characterised by their non–functional requirements, such as limited system resources or the extreme environment they operate in. How can we best define the software architecture of a system that is both real–time, embedded and distributed? After our own experiences with VDM in other projects and inspired by [1], we selected VDM++ in combination with the Unified Modeling Language (UML) as our tools for this project. The VDM++ language [10] provides a good mix between the ability to specify temporal and functional properties within one paradigm. The real–time Unified Modeling Language [5] provides a graphic framework to better communicate the structure and timing requirements of the design. During the specification phase the IFAD VDM++ toolkit, Rational Rose and especially the Rose–VDM++ coupling were used extensively.

We shall report on our experiences by providing our comments to three well–known publications on the do's and don'ts of formal methods. We will use the paper "Ten Commandments of Formal Methods" [4] to structure the discussion.

Commandment I : Thou shalt choose an appropriate notation

There is a wide range of formal notations available nowadays. They all excel in some way, be it specification of temporal or behavioural aspects, tool support or executability. How do you select the language that is most appropriate for your project?

The "Ten Commandments" paper proposes that the characteristics of the specification language is the main discriminating factor for this selection. These language characteristics should then be matched against the characteristics of the problem domain. In an industrial context this choice is certainly not an obvious task. The decision has to be made in a short period of time at the start of the project. Sometimes, it is even the deciding factor whether or not a client is going to hire us, so the pressure is on, right from the start. At first glance, one would expect the real–time characteristics of the clock system to be the main problem to tackle, especially because this was also confirmed (and emphasised) by our client at project start–up. We have learned that our customers are often biased towards either functional properties or temporal properties that the system should adhere to and that this does not provide a sound basis to select the appropriate notation. It is our role as an IT service supplier to investigate whether this problem statement is indeed the core of the problem and this turned out *not* to be the case. After some preliminary investigations, we determined that in this case describing the functional properties of the system was much more important than the temporal aspects. We consider this mismatch to be a common pitfall.

The second observation made is that the balance between the technical and non–technical aspects driving the notation choice is much more in favour of the non–technical aspects than proposed by the "Ten Commandments" paper. A common fact, that is easily overlooked, is the accepted way of working at the client. Often, a strong preference for methods and standards exists (including local dialects) and it is very difficult to persuade the client to accept new methods or techniques. The only way to overcome this problem is to show the client that formal methods can be easily integrated in their way of working. Furthermore, clients wish to comply with the mainstream of IT development techniques, partly to be independent from their IT suppliers and service providers, and partly to be "politically correct". The fact that VDM++ could be easily integrated into the defacto industry standard UML practically removed the acceptance question from our agenda. The availability of good tools and an ISO standard for VDM silenced the remaining opponents.

With respect to the myths papers [9] and [3], we can conclude that

- myth 3 (*Formal Methods are only useful for safety–critical systems*)
- myth 9 (*Formal Methods lack tools*)
- myth 12 (*Formal Methods are unnecessary*)
- myth 13 (*Formal Methods are not supported*)

are indeed myths.

Commandment II : Thou shalt formalize but not overformalize

Essentially, the issues raised by this second commandment are when and where to apply formal methods and to what level. Our client has been developing mainly administrative applications. The design methods that are standardised in the organisation are thus fully based on this system class, and are not appropriate for developing a real–time distributed system. Hence there was a clear need to define a new set of tools to address our problem.

The client commanded the use of object oriented technology, hence the choice for the industry standard Unified Modeling Language (UML) and design patterns [6] was easily made. However, we also determined that the level of detail provided by UML (and its associated object constraint language, OCL) was not sufficient for our purpose. In UML, it is possible to define classes and relations between classes. These relations can be formalised using the OCL, but the OCL is unsuited for the specification of complex algorithms within class member functions, due to the restricted languages syntax and semantics. UML merely provides a placeholder for a natural language specification of the algorithm (e.g. using pseudo code). We selected VDM++ to compensate for this weakness. A VDM++ specification is inserted into the UML model, instead of a natural language specification.

However, one must be very careful to introduce this type of technology into any organisation. In general, we believe that the maturity of the client's organisation is leading in determining the level to which formal methods can be applied. It should be easy to fit this technology into their current way of working, a clear transition path must be available (no giant leap for mankind). In our case, just using formal specification was already a major step forward. If we would have aimed at a higher level of formalisation, we are convinced that we would have failed to achieve our goals. This is also confirmed by [7].

To help the acceptance of the formal specification by the client, we intentionally kept the level of abstraction of the specification as low as possible, without sacrificing specification quality. Aided by the UML diagrams, programmers with virtually no experience in formal methods were capable to read and understand the specification. Even more, in their opinion, the VDM++ specifications were of considerable added value to better understand the UML diagrams. Hence we can confirm that it is not necessary to have highly trained mathematicians to use formal methods (Myth 4 in [9]).

One of the obvious benefits of the use of formal methods is to make requirements clear and precise in the early phases of the project. This reduces the risk that a system is not in line with the client's needs. In contrast, a more precise specification also reveals the things that you specifically *do not* build. The client likes the risk reduction aspect of this approach but his elbowroom, to adapt to moving or unclear end–goals, is much more limited than using traditional, informal methods. What is clearly recognised as a requirements change when using formal methods, was not always recognised as such using informal methods. This can be threatening to the client since his own capabilities have been proven to be insufficient to prevent this change in the first place. Furthermore, he is more often faced with (e.g. financial) consequences of these changes, even though the overall project cost may still be considerably less than by using informal methods. This requires a new way of working for the client and he should be open to this.

We have shown that in our project:

- myth 2 (*Formal Methods are all about program proving*),
- myth 4 (*Formal Methods require highly trained mathematicians*) and
- myth 6 (*Formal Methods are unacceptable to users*)

do not apply.

Commandment III : Thou shalt estimate costs

This commandment is a generic claim, which is not very specific to the use of formal methods. However, the introduction of formal methods influences the amount of time that is spent in each of the project phases. In general, more effort is spent in the earlier phases of the project, compared to traditional software development. Due to the clearer specifications, this time can than be regained during the implementation and testing phases. Nevertheless, clients often are reluctant to accept this shift in time spending, mainly due to the fact that the deliverables produced (a "dead" specification document) basically remain the same on the outside. It is therefore essential to capitalise on the inherently higher quality of the specification document. The client can be given "value for money" by using an iterative specification approach (such as [11]), supported by executable specifications. The specification process can be steered using techniques from Rapid Application Development such as the Dynamic System Development Method (DSDM), including interactive workshops, multiple deliverables, time boxing and prioritizing requirements.

The gradual introduction of formal methods can limit the costs associated with training, consultancy and support tools. We adapted the specification level to minimise the need for training. We could suffice with on–the–job training and providing references to good textbook material [8]. To verify that the level of competence of the team members was high enough, a peer–to–peer review was performed on the design by the client's implementation team. Using traditional methods and techniques, the consultancy costs would have been at least as high as with formal methods in this project.

The main achievement of the specification effort was that the implementation team was capable of producing a detailed planning of the implementation phase. Moreover, the actual implementation has been performed exactly according to that planning.

We have shown in this paragraph that:

- myth 5 (*Formal Methods increase the cost of development*) and
- myth 8 (*Formal Methods delay the development process*)

do not necessarily apply.

Commandment IV : Thou shalt have a formal methods guru on call

In an industrial environment it is essential to determine project risks. Weaknesses should be compensated for, strengths should be exploited. Introduction and use of any new method relies on having someone that knows the ropes. Formal Methods are no exception to this rule. We agree with Bowen and Hinchey that either training or hiring expertise from outside the company can compensate for a lack in formal methods knowledge.

As noted earlier, it is not the core business of our client to develop real–time embedded software systems using formal methods, therefore they decided to hire the expertise from outside and do some on–the–job training.

Commandment V : Thou shall not abandon thy traditional development methods

We choose for a hybrid approach using UML and VDM++. UML is especially suitable for presenting the system structure and visualizing the system behaviour using sequence diagrams. On the other hand, UML is not well suited for expressing functional details. Specification stops at the function interface level, providing only natural language to explain what the function should do. That is where VDM++ hooks in, it allows for concise specification of all system details.

The IFAD VDM++ and Rational Rose tools can be used for round–trip engineering and we have used this facility extensively. Often, the structure of the application was first written in VDM++ and converted into UML. After specifying sequence diagrams in UML, often extra operations had to be defined that could be pushed back into the VDM++ specification.

Notwithstanding the good tool support, the design problem did not get any less complicated. To solve the specification problem, we did two things. First, we defined a clean interface layer on top of the target operating system (VxWorks), that abstracted away from the operating system specific functions for handling message queues, semaphores, sockets and the multi–processor features of the system. Secondly, we used design patterns extensively to structure the application. Design patterns helped us to break the design problem in manageable pieces. After selecting the most appropriate design pattern for a design problem, a VDM++ and UML specification of that part of the application was written in a round-trip style,, as outlined above.

The design was hand–coded in C++, directly from the specification. Due to the operating system interface layer that we defined, it was possible to test the code already on the host platform (running Windows NT). Since the application of C++ in embedded systems is already considered a quantum leap forward, we did not attempt to use the IFAD code generator.

In this paragraph we tried to substantiate the claim that:

- myth 6 (*Formal Methods are unacceptable to users*) and
- myth 10 (*Formal Methods replace traditional engineering design methods*)

do not hold.

Commandment VI : Thou Shalt Document Sufficiently

For industrial applications, building an application is only the beginning of a much longer maintenance life cycle. In our case, the expected life time of the application is 15 years. If the usage of formal methods would be sufficient in the maintenance life time of the application, no additional documentation would be needed. However, formal specifications are only a part of the description of a system. All kinds of aspects like business strategy, system context, overall design decisions, systems architecture, software architecture, etc. are still difficult to express using formal methods, but are necessary for the maintenance of an application. Therefore, we agree with Hinchey and Bowen that extra care should be taken with respect to documentation.

The commandment to document sufficiently is more a statement that the usage of formal methods does not replaces the traditional form of documentation but extents it in a way that reduces ambiguity and errors. In our case this means that the amount of documentation is expected to be at least the same as in traditional documentation standards but the level of substance is larger.

The combination of formal methods and non–formal methods (natural language) also creates the problem of consistency between these two parts and they are of course not automatically checkable. The only guarantee to keep these parts consistent is by reviewing, which is in fact the only method available to keep non–formal documentation consistent. Tool integration could help in making documentation updates easier. In developing the auction system we used the IFAD VDM++ tools in combination with Rational Rose. Both tools are highly integrated with each other and with Microsoft Word. This set of tools made it possible to promote changes in the VDM++ specification to the UML diagrams and vice versa, directly from the overall specification document in Word. In conjunction with a version management tool such as Microsoft Source Safe or Rational ClearCase, this provides sufficient support for document management.

Commandment VII : Thou Shalt not Compromise Thy Quality Standards

Having good tools and well–trained people still does not guarantee a good product. Formally specifying a product the client does not want is still possible. As with any method, formal methods supply a means and are not a goal in itself. We totally agree with the commandment not to compromise quality standards and to integrate the usage of formal methods in these standards.

However, we had to deal with an organization at CMM level 2 which means that there was no formal quality system available for system development (this is an CMM level 3 activity). From our perspective, the usage of formal methods is just a part of good software engineering practice, so we introduced IEEE–12207[1] based deliverables and review techniques in this project to make the usage of formal methods more effective. As a result, the team members started to demand the same level of quality from collegues in other parts of the project, effectively implementing an ad–hoc quality standard.

This commandment contradicts

– myth 1 (*Formal Methods can guarantee that software is perfect*) and
– myth 10 (*Formal Methods replaces traditional engineering design methods*).

Commandment VIII : Thou shalt not be dogmatic

Our starting point as IT service provider is an industrial dogma: make what the client wants as soon as possible and at reasonable cost. We will use all techniques that can help us to achieve this goal. From that perspective, formal methods is just yet another technique. If it is an advantage, use it, otherwise don't. Sometimes better methods are available, for example when describing user interfaces.

Also, non technical aspects can influence the degree of freedom we have, for example in using client preferred tools or techniques and the expected resistance towards formal methods. In our case that meant that no formal methods have been used to develop the network infrastructure (an equally important part of the system) and user interface parts of the client software.

This commandment contradicts

– myth 3 (*Formal Methods are only useful for safety–critical systems*) and
– myth 14 (*Formal Methods people always use formal methods*).

Commandment IX : Thou shalt test, test, and test again

Testing remains by far the most important phase in the development process. It gives the possibility to check that the design matches the client demands (validation, did we build the right product) and to check whether the design operates as intended (verification, did we build the product right). Especially in the area of real–time embedded applications this is a very important phase.

In an industrial project it is very likely that commercial–off–the–shelf products will be used. These products are pushed to the limits of their capabilities or applied in a slightly non–standard manner. Their operation is already tested at the factory. We work from the specifications of the system. From a testing point of view these products can thus be regarded as black boxes. To some extend their operation can even be simulated, providing an easy way to perform module testing. Nevertheless, even if the module tests have been successful, when these systems are integrated we still find errors. These errors are often difficult to find since they are mostly not related to the system itself but to the environment in which the system operates. Examples of such errors are: electric incompatibility of components, signal interference, corrupt hardware etcetera. It is also not unlikely that you will find errors in the commercial–off–the–shelf products.

Formal Methods can not prevent these errors from occurring. The amount of things that can go wrong is simply far to great to model. Nevertheless, formal methods give you the tools to

[1] Formally known as MIL–STD–498, which was the successor to DOD–2167a

specify a system that is robust and fault–tolerant, such that it is able to detect contingency situations. Furthermore, formal methods provide an excellent framework for testing. Apart from the executability issue, the specification itself can be used to deduce test cases. Due to the detail and clarity of the specification, it is often much easier to determine the cause of the problem than using informal methods.

Obviously, this contradicts

– myth 1 (*Formal Methods can guarantee that software is perfect*).

Commandment X : Thou Shalt Reuse

Reuse sounds much better than development from scratch, because development normally costs more time (and money) than reuse of existing solutions. Most of the newly introduced techniques promise that reuse is much easier if this particular technique used. Hinchey and Bowen suggest that formal methods will be an enabling factor for reuse.

However, our experiences with object oriented technology (for which the same promise was made) and formal methods is that reuse is still difficult to achieve. Especially in the area of technical automation, systems are often developed in very low volume in a highly specific application area which makes the extra investment needed to develop reusable components not worthwhile.

In our opinion, reuse only works for non–typical application aspects such as standards and generic knowledge about a problem domain (design patterns). We believe that the best level of reuse that can be achieved is evaluation of experiences gained from performing projects.

Lessons learned and conclusions

We have applied VDM++ and real–time UML very successfully in our project. Our client is going to continue to work with this development approach for future extensions to the auction clock system. We have formulated several lessons learned from our experiences:

– During method selection, take your time to investigate the core problem. Selection of the wrong tool can get you in real trouble later (too late).
– Be aware that the balance between technical and non–technical issues driving the notation choice are often in favour of the non–technical aspects.
– Determine the maturity level (and flexibility towards change) of the organization. Adapt the level of formality accordingly, when introducing this kind of technology.
– Make clear right from the start that more time is spend during the system specification phase. Use an iterative design approach to show progress and gain client commitment.
– Embed the formal method of your choise in the current way of working using traditional tools. If a software development quality standard does not exist, make sure the basic review processes are put in place. Otherwise, the software development process itself will become the weakest element in the project chain.
– Test, test and test again!

We have discussed the “Ten Commandments of Formal Methods” in the context of our project. We disagree with commandment 10 (*Thou shalt reuse*). We have been able to show that 13 out of 14 myths were indeed myths, in our particular case. Myth 11 (*Formal Methods only apply to software*) was out of context for this project, but nevertheless, it has been proven to be a myth more than once by others (see for example the Formal Methods application database at `http://www.fme-nl.org/`).

References

1. S. Agerholm and W. Schafer. Analyzing SAFER using UML and VDM++. April 1998.
2. J.P. Bowen and M.G. Hinchey, editors. *High–Integrity System Specification and Design.* FACIT series. Springer Verlag, April 1999.
3. J.P. Bowen and M.G. Hinchey. *Seven More Myths of Formal Methods*, pages 153 – 166. In *FACIT series* [2], April 1999. Originally published in *IEEE Software*, july 1995.
4. J.P. Bowen and M.G. Hinchey. *Ten Commandments of Formal Methods*, pages 217 – 230. In *FACIT series* [2], April 1999. Originally published in *IEEE Computer*, april 1995.
5. Bruce Powell Douglass. *Real–Time UML, Developing Efficient Objects for Embedded Systems.* Addison–Wesley, Object Technology Series, 1998.
6. F. Buschmann et al. *Pattern–oriented Software Architectures: A System of Patterns.* John Wiley & Sons, 1996.
7. W.J. Brown et al. *Anti–patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998.
8. J. Fitzgerald and P.G. Larsen. *Modelling Systems, Practical Tools and Techniques in Software Development.* Cambridge University Press, 1998.
9. J.A. Hall. *Seven Myths of Formal Methods*, pages 135 – 152. In Bowen and Hinchey [2], April 1999. Originally published in *IEEE Software*, september 1990.
10. The Institute for Applied Computer Science. *VDMTools: The IFAD VDM++ Language*, 1998.
11. P. Kruchten. *The Rational Unified Process: An Introduction.* Addison–Wesley, Object Technology Series, 1998.

Acknowledgements

The authors wish to thank Bert Bos, Max Geerling and Peter Gorm Larsen for their comments on early versions of this paper.