USING EXECUTABLE VDM++ MODELS IN AN INDUSTRIAL APPLICATION
SELF-DEFENSE SYSTEM FOR FIGHTER AIRCRAFT

Electrical and Computer Engineering
Technical Report ECE-TR-1

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: Using Executable VDM++ Models in an Industrial Application - Self-defense System for Fighter Aircraft
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-1

Author: Sune Wolff
Department of Engineering – Electrical and Computer Engineering, Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: **http**://www.eng.au.dk

Year of publication: 2012 Pages: 15
Editing completed: October 2011

Abstract: When developing complex software systems, one of the most significant challenges is to make sure that the customer and developer agree on the requirements of the system. By using executable models early in the development process, a higher degree of confidence can be gained in the system design and misunderstandings or ambiguous functional requirements can be avoided. This paper presents an industrial case of a communication protocol between two parts of a self-defense system used on-board fighter aircraft. An executable model of both systems were created using the Vienna Development Method (VDM), and exercised using many scenarios to cover different corner cases. This was done as an alternative to analysing all the scenarios by hand, which would be much more time consuming and far more error prone. The results of the scenario based tests were used to communicate with the customer and ensure that agreement of the requirements was reached.

Keywords: VDM, industrial application, defense, self-defense, fighter aircraft

Supervisor: Peter Gorm Larsen
Financial support: Danish Agency for Science Technology and Innovation

Please cite as: Sune Wolff, 2012. Using Executable VDM++ Models in an Industrial Application - Self-defense System for Fighter Aircraft. Department of Engineering, Aarhus University. Denmark. 15 pp. – Technical report ECE-TR-1

Cover photo: Official U.S. Air Force's photostream on Flickr

ISSN: 2245-2087

Using Executable VDM++ Models in an Industrial Application: Self-defense System for Fighter Aircraft

Sune Wolff [1,2]

[1] Aarhus School of Engineering, Denmark, swo@iha.dk
[2] Terma A/S, Denmark, sw@terma.com

Abstract. When developing complex software systems, one of the most significant challenges is to make sure that the customer and developer agree on the requirements of the system. By using executable models early in the development process, a higher degree of confidence can be gained in the system design and misunderstandings or ambiguous functional requirements can be avoided. This paper presents an industrial case of a communication protocol between two parts of a self-defense system used on-board fighter aircraft. An executable model of both systems were created using the Vienna Development Method (VDM), and exercised using many scenarios to cover different corner cases. This was done as an alternative to analysing all the scenarios by hand, which would be much more time consuming and far more error prone. The results of the scenario based tests were used to communicate with the customer and ensure that agreement of the requirements was reached.

Keywords: VDM, industrial application, fighter aircraft, requirements, executable model, lightweight formal analysis

1 Introduction

When developing complex software systems, one of the most significant challenges is the communication between the customer and the software engineers implementing the system. The customer is the domain expert who possesses detailed knowledge of the domain in which the system will operate, and it is imperative that the software engineer obtains the knowledge needed to solve the task at hand. Missing or ambiguous systems requirements is a very common cause of project delays since customer and developer do not share the same view of the system under development. Errors in communication can lead to crucial defects in the final system – something that must be avoided at all costs in a safety-critical system like the one presented here.

The most commonly used means of communication is natural language, which unfortunately brings the potential for errors such as subtle misunderstandings and ambiguously described functionality. To solve these issues, formal methods have been applied to several industrial cases over the last 20 years [28]. By utilising formally described system models, important details of the system can be described rigorously while a higher level of abstraction can be applied to less important details of the system.

Executable subsets of many formal specification languages exists, enabling the developer to show the customer running scenarios of the proposed system – this is a great

way of communicating the technical solution to a less-technical adept customer and agree on the systems design. The use of executable system specification has been subject to much debate in the past [10,8]. One of the goals of the case study presented here, was to examine the benefits of creating an executable formal model specifying the functional behaviour of the system and using lightweight formal methods analysis principles to gain insight in the system-level properties.

In the software industry, the Unified Modeling Language (UML) [19] has seen wide usage, as a means to provide an architectural and functional overview of the software. UML models can even be executable, allowing domain experts to do scenario based tests of the model – unfortunately, this requires very refined models which are very close to the final implementation, hence removing the advantage of using abstraction in the modelling phase. Abstraction is a key element – not only when applying lightweight formal methods analysis as described in this case study – but in software development in general [12].

In order to evaluate the use of lightweight analysis techniques, an executable model of a self-defense system used on-board fighter aircraft was developed using the formal method Vienna Development Method (VDM) [3]. Using this approach has the advantage of providing executable models while maintaining the ability to apply a higher level of abstraction to details not needed to describe the functionality of the system. This system level model was used as a means of communication between domain experts and the group of software engineers implementing the system, when developing an expansion to a communication protocol between two subsystems. The case study presented here is part of the Industrial PhD project described in [27].

Initially, an overview of the formal method VDM as well as the different tool support available is given in Section 2. An overview of the self-defense system used as a case study is given in Section 3, followed by a description of the VDM model and the test setup in Section 4. The results obtained from the study are presented in Section 5, followed by concluding remarks as well as a description of future work in Section 6.

2 VDM Modelling Languages and Tool Support

VDM is a collection of techniques to formally specify and develop software. VDM originates from IBM's Laboratories in Vienna in the 1970s. The very first language supported by the VDM method was called Vienna Definition Language (VDL), which evolved into the specification language VDM-SL [4] which has been ISO standardised [11]. Over the years, extensions have been defined to model object-orientation (VDM++ [5]) and distributed real-time systems (VDM-RT [18,24]). Two alternative tools exists; the commercial tool VDMTools [6] and the open source initiative Overture [20].

Data in VDM models can be described using simple abstract data types such as natural numbers, booleans and characters, as well as product and unions types and collection types such as sets, sequences and mappings. The system state can be described using `state` in VDM-SL and `instance variables` in VDM++ and VDM-RT, the value of which can be restricted by invariants. To modify the state of the system, operations can be defined either explicitly by imperative statements, or implicitly by

pre- and post-conditions. Functions which cannot use or modify the state can be defined in a similar fashion.

VDM has been used in several successful industrial applications e.g. [15] – examples of two recent applications in the Japanese industry is the TradeOne system developed by CSK systems for the Japanese stock exchange [5] and the FeliCa contactless chip firmware [14,13]. Most of these applications have the common goal of providing rapid feedback on requirements and design early in the development cycle, just like the project described here.

The purpose of the model was to describe the system functionality and test a multitude of combinations of system state and input, and not to test synchronization of a concurrent system. Hence, for the case study presented here the sequential version of VDM++ was chosen, to be able to describe an object-oriented architecture of the system without introducing the added complexity of a concurrent model. In order to permit the use of scenario-based tests, only the executable subset of VDM was used. Further more, even though some implicit functions and operations can actually be executed (see [7]), the model only makes use of explicitly defined functions and operations.

3 Industrial Case: ECAP

This section gives a functional overview of the self-defense system as well as the protocol which is the main focus of this study. There are details of the system which cannot be presented here due to military classification restrictions, but hopefully sufficient information is given to introduce the reader to this complex system.

3.1 Functional Overview

When fighter pilots are flying missions in hostile territory, there is a risk of encountering enemy anti-aircraft systems. To respond to these threats, the pilot can deploy different countermeasures. Since opposing anti aircraft systems are becoming increasingly sophisticated, and on-board self-defense systems are also becoming more sophisticated, the fighter pilot is in need of assistance in choosing the optimal countermeasures strategy.

A system called *Electronic Combat Adaptive Processor* (ECAP) has been developed to assist the pilot in choosing the most optimal response to incoming threats. The system is a programmable unit that provides threat adaptive processing, threat evaluation and countermeasures strategy to counter incoming threats. From a multitude of sensor inputs (aircraft position, orientation, speed, altitude and threat type and incoming angle to name a few) the system chooses an effective combination of countermeasures against the incoming threat. The different sensors attached to ECAP can detect different types of threats, and will report data of any incoming threat of that specific type to ECAP. The chosen threat response, which can consist of one or more countermeasure programs, is sent to a Dispenser subsystem which administers the deployment of the correct types of dispense payloads with the correct timing. An overview of the system can be seen in Fig. 1.

The system can operate in two different modes; semi-automatic and automatic. When the system is running in automatic mode, all responses are carried out without further delay, but when the system is operating in semi-automatic mode, the pilot needs to consent to all generated responses. When a response is generated, it is placed in a queue and the pilot is notified. Once a consent from the pilot is received, the response is sent to the Dispenser and removed from the queue. If multiple responses are placed in the queue, these are sorted based on the priority of the corresponding threat.

In order to allow the dispensed countermeasures to have the expected effect, a Reassessment Timer (RT) keeps track of when to reassess the threat. A similar precaution against obscuring dispensed countermeasures is an Interference Avoidance Timer (IAT) which specifies a period of time in which a specific type of countermeasure cannot be used in another threat response. This ensures that unwanted interference with the current countermeasures program, which potentially could cancel the effect, is avoided.

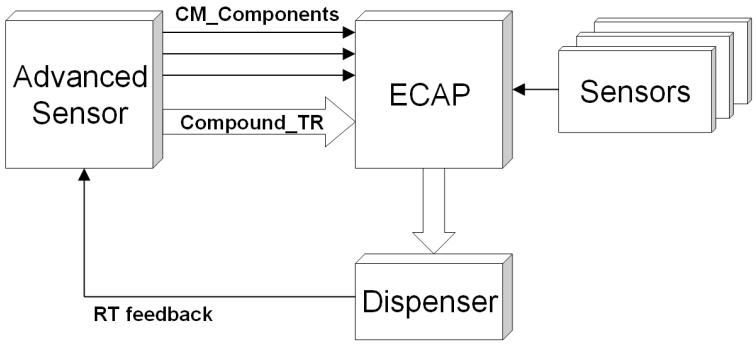

Fig. 1. Self-defense system overview

The subsystem of interest for this paper, is a special Advanced Sensor (AS). This sensor not only detects incoming threats, but also calculates the countermeasures needed to avoid the threat. AS is running in parallel with the rest of the system, and relies on ECAP to accept and execute generated threat responses. Hence, ECAP needs to check the RT and IAT of the proposed response for conflicts, and accept/reject the response based on this. A robust protocol has been specified, defining the communication between ECAP and AS. Initially, ECAP treated the sequence of components which a threat response consists of separately which resulted in the need for several pilot consent actions in order to execute a response when the system was running in semi-automatic mode. Not only did this put unnecessary strain on the pilot, but it also resulted in delays between the different countermeasure programs.

The main focus of this case study was an update to the way ECAP interprets messages from the AS system. The protocol itself has undergone military certification, hence no changes to the protocol were possible, since this would involve re-certification of the protocol, which is both a costly and time consuming task. Instead, in addition to the individual components of the threat response, AS will also generate a compound

threat response message which is the concatenation of the sequence of components. The only thing distinguishing a component from the compound threat response is the position in the complete AS message – in a message of length = n, all sub-messages [1..n-1] are components and the n'th message is the compound threat response as shown in Fig. 2. For example, the first component could be a dispense routine using one type of payload, the second component could be a command to a subsystem – then the compound message would consist of both the dispense routine defined in the first component and the command defined in the second component.

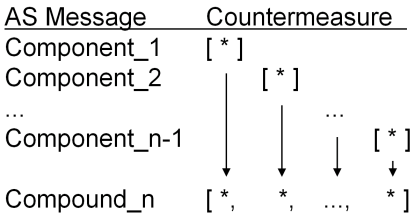

Fig. 2. Countermeasure components and compound threat response from AS

ECAP will still test for conflicting RT and IAT for all the components, but will only ever execute a compound threat response. This ensures that only a single pilot consent is ever needed, which in turn also solves the issue of unwanted delays between countermeasure programs. At any time before the accept of the final complete threat response, ECAP can cancel the requests from AS if another threat surpasses the AS request, or if a higher priority threat is discovered by another subsystem. The use of components ensures that AS knows exactly which part of the compound message is rejected by ECAP, and it will then try to generate another effective compound threat response not using that particular component.

The combination of RT, IAT and different levels of priority of threats coming from different subsystems, makes the functionality of, and communication between ECAP and AS very complex. In order to gain confidence in the proposed update, a lot of corner cases needed to be analysed to ensure no intricate details had been ignored. Traditionally, this analysis would have been done by hand which is a very time consuming and error prone process. As an alternative, an executable model of the entire updated ECAP system with the AS subsystem was developed and analysed using the VDM interpreter from Overture.

3.2 Project Setup

The expansion of ECAP described above was only a small part of a larger update to the self defense system used by the customer, including expansions to other parts of the on-board self defense system – but only the expansion to ECAP was subject to the research described in this paper. Traditional software development methods (mainly Scrum [23] based on a thoroughly negotiated backlog) were used to develop the system.

The project had a very short timespan with only four months of development time with a workforce of roughly 30 engineers. Under normal circumstances, modelling is done prior to development, but because of the short timespan of the project, the modelling work was done in parallel with the development of the remaining parts of the system by a single person (the author of this paper). This was possible since the update to the protocol interpretation was one of the last tasks of the project, which gave enough time to finish the large task of modelling all of the ECAP system as well as the AS functionality and the existing protocol.

4 VDM model of ECAP

This section gives an overview of selected parts of the model of the ECAP system with the AS, using the ASCII notation of VDM. Additional sensors are also included in the complete model, but they are omitted here for reading convenience.

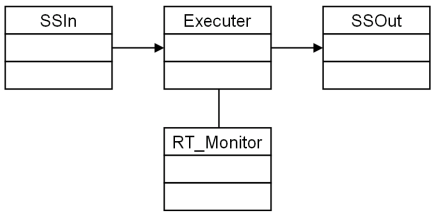

Fig. 3. Class diagram of ECAP

In Fig. 3 a simplified class diagram of ECAP is given, only including the few classes which will be presented below. The `SSIn` class manages all messages sent from a subsystem to ECAP including all messages from AS. The `Executer` interprets all the messages received, generates the most optimal threat response and tests for conflicts using the `RT_Monitor` class. All commands and messages from ECAP are sent via `SSOut` to the correct subsystem, again including the AS subsystem.

4.1 System Modelling

A complete request from the AS subsystem includes all the individual countermeasure components and the compound threat response which is a combination of the individual components, as seen in Fig. 2. Each individual component only consists of an ID and a message type tying it to the protocol.

Internally in ECAP, the ID of a message is mapped to a specific countermeasure – each specifies if a pilot consent is needed before execution, the different dispense programs engineered to counter the given threat as well as the RT and IAT. Information like priority of the threat and optional subsystem commands are omitted in the following description.

All commonly used types in the model are defined in the ECAP_Types class, which all other classes inherits from. This ensures that all parts of the system has the same understanding of message structures etc. This class in omitted in Fig. 3 for reading convenience.

```
class ECAP_Types

types
  public Countermeasure ::
    Consent  : bool
    DispProg : Program
    IAT      : IA_TimerStruct
    RT       : nat
  inv cm ==
    (cm.DispProg.Type1 <> [] => cm.IAT.Type1 > 0) and
    (cm.DispProg.Type2 <> [] => cm.IAT.Type2 > 0);
```

The invariant specifies that if a certain type of dispense payload is used in the dispense program, the duration of the IAT for this type of payload must be greater than zero. The dispense program DispProg consists of a sequence of different dispense payload types. Between each element of these sequences, a variable delay is added. This enables the Electronic Warfare Programmer, who configures the self-defense system, to engineer the most optimal dispense pattern to counter the threat. Either of these sequences can be empty if the specific type of dispense payload is not needed to counter the incoming threat.

```
  public Program ::
    Type1 : Routine
    Type2 : Routine;

  public Routine = seq of RoutineStep;
  public RoutineStep = nat * DispensePayload;

  public DispensePayload = <Type1> | <Type2>;
```

The final part of the threat response is the definition of IAT and RT. The reassessment timer RT is just a natural number specifying how long the period is, whereas a record type is used to define IAT, with fields for each of the types of dispense payloads. This ensures that even though there is IAT on one type of payloads, ECAP can still respond to a threat with a dispense program using another type of payload.

```
  public IA_TimerStruct ::
    Type1 : nat
    Type2 : nat;
```

Since the protocol between AS and ECAP was locked for change, only the way ECAP interprets messages sent from AS was changed as described in Section 3. All

incoming messages from the AS subsystem are passed to IncomingAS_Message in the Executer class.

```
class Executer is subclass of ECAP_Types

instance variables
  private AS_OutMsg : seq of AS_Msg := {};

operations
  public IncomingAS_Message : AS_Msg ==> ()
  IncomingAS_Message (as_msg) ==
   (for all component in set elems as_msg
    do
     (HandleComponent(component);
      ...);
    SSout.AS_output(AS_OutMsg);
    ...);
```

Each of the countermeasure components in the AS message is processed by the Executer in the operation HandleComponent. One of the tasks of this operation is to check for conflicting RT. Once all components have been processed as described in the protocol, a new AS_Msg has been generated consisting of responses to the AS subsystem. This message is passed to the AS subsystem using the operation AS_output in the SSout class representing all communication from ECAP to the attached subsystems.

For each component the corresponding countermeasure is generated using a table, mapping the ID to the countermeasure. Following this, the Executer checks for conflicting IAT and makes sure that there are enough dispense payloads left to execute the countermeasure program. This is done in the TestDispenseAndIAT operation, which appends a message to the output message to AS if these constraints are not fulfilled.

```
  private AS_ComponentReceived : Component ==> ()
  AS_ComponentReceived (c) ==
    let cm : Countermeasure =
             AS_RespTable.ResponseTableLookup(c.ID)
    in
      if TestDispenseAndIAT(cm)
      then AS_OutMsg := AS_OutMsg ^ [mk_AS_Msg(c.ID)];
```

Once the negotiation defined in the protocol results in a confirmation of the proposed compound threat response, the response is executed using the operation Execute-Response. ECAP will never execute a countermeasure component, but only the concatenated compound threat response.

```
  private ExecuteResponse : Countermeasure ==> ()
  ExecuteResponse (cm) ==
    if cm.Consent
```

```
    then ECAP`Queue.AddCountermeasure(cm)
    else (SSout.ExecuteCountermeasure(cm);
          ECAP`RT.AddRT(cm);
          ECAP`IAT.AddIAT(cm));
```

If ECAP is operating in semi-automatic mode a pilot consent is needed before executing the response – hence it is placed in a queue waiting for the consent. In full-automatic mode the response is executed directly by sending it to the Dispenser subsystem and IAT and RT is added as specified in the countermeasure.

```
class RT_Monitor is subclass of ECAP_Types

instance variables
  private RT_Map : map nat to nat := { |-> };

operations
  public AddRT : Countermeasure ==> ()
  AddRT (cm) ==
    let t : nat = World`timerRef.GetTime()
    in
      RT_Map := RT_Map munion {cm.ID |-> t + cm.RT}
  pre cm.ID not in set dom RT_Map;
```

The class RT_Monitor contains a mapping of response ID to their RT deadline. When RT is added to a threat response using the AddRT operation, the ID is mapped to the current time plus the RT period specified in the response. Time is simulated in a custom built timer class, and is represented by a counter which is incremented periodically. A precondition ensures that RT has not yet been specified for the given response. Similar functionality is specified for IAT.

Periodically the RT_Monitor analyses this map, and removes any thread ID for which the deadline is up.

```
  private PeriodicOp : () ==> ()
  PeriodicOp () ==
    let t : nat = World`timerRef.GetTime()
    in
      for all id in set dom RT_Map
      do
        if RT_Map(id).Deadline <= t
        then DeadlineIsUp(id);

  private DeadlineIsUp : nat ==> ()
  DeadlineIsUp (id) ==
   (if RT_Map(id).SS_Type = <AS>
    then ECAP`Exe.AS_RT_Done(id);
    RT_Map := {id} <-: RT_Map;
   );
```

In addition to removing the response ID from the map when the deadline is up, the Executer is notified when any AS generated response is removed, since according to the protocol AS needs to be notified when RT for a response has finished. This is done in the `AS_RT_Done` operation in the Executer class, which generates a message to the AS subsystem.

```
class Executer is subclass of ECAP_Types

operations
  public AS_RT_Done : nat ==> ()
  AS_RT_Done (id) ==
    SSout.AS_output([mk_AS_Msg(id)]);
```

4.2 Testing The Protocol

In order to test the new interpretation of messages from AS, a unit test framework called VDMUnit was used (described in [5]). This framework is inspired by JUnit [2] and can be used to set up test suites for unit and integration tests.

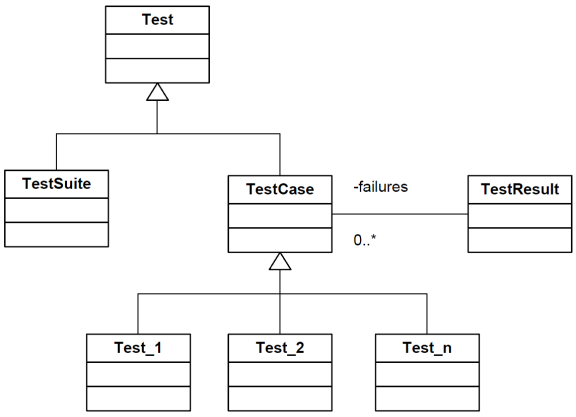

Fig. 4. Class diagram of VDMUnit

A `TestSuite` includes a sequence of instances of the class `Test`. Usually, these instances are `TestCases`, but it is also possible to have a test suite containing other test suites in order to create a hierarchical structure of the different unit and integration tests. `TestResult` keeps a list of failed tests, which are displayed after all tests in the test suite have executed.

The model can be exercised using public operations and functions. By using the special operations `AssertTrue` and `AssertFalse` built into VDMUnit the model state can be tested against the expected state. More than 20 different scenarios were

constructed, testing a lot of different combinations of ECAP system state, IAT, RT and requests from the AS subsystem.

```
class SimpleTest is subclass of TestCase

instance variables
  private world : World:= new World();

operations
  protected Test: () ==> ()
  Test() ==
   (AssertTrue(World`timerRef.GetTime() = 0;
    World`timerRef.IncTime();
    AssertTrue(World`timerRef.GetTime() = 1);
    ...
   );
```

In addition to these scenario-based tests, the VDM model was examined using the *Proof Obligation Generator* [22] built into the Overture tool. The tool automatically generates all the proof obligations which have to be discharged in order to guarantee the internal consistency of a model. Every time a partial operation is used a proof obligation is generated flagging a potential runtime error. For example, every time the head operator returning the first element of a sequence is used, the model designer must ensure that the sequence in not empty which would result in a undefined return value.

5 Results

In total, the complete model of ECAP and the AS subsystem consists of more than 1800 lines of VDM++ specification. In addition, more than 1500 lines of test were created to run the many scenarios needed to exercise the new use of the protocol. Built into the Overture tool is the ability to generate test coverage of a model, which gives an indication of parts of the model which are exercised less than other parts. The AP subsystem has a test coverage of 100%, meaning that every line of specification has been exercised by the scenarios. On average, the complete model of ECAP, AP and all other subsystems has a test coverage of 94.1%. The focus of the scenarios was on testing the new interpretation of the protocol, and testing combinations of ECAP system state with different AP input. This is the reason why every branch of the complete model has not been completely covered by the tests, but only the parts of the system concerned with the communication between ECAP and AS.

The ECAP and AS model made use of extensive logging, so at any point in time the system state was available for post-execution analysis. The logfiles from the many scenarios were used directly in the communication with the customer, to give a precise description of how the systems should react in the different situations. This was a great aid in agreeing on the way ECAP should interpret the countermeasure components and compound threat response. In addition to these logs built directly into the model, a certain amount of logging is available in the Overture tool. The main focus

of these automatically created logs is multi-threaded models, so mainly the scheduling of threads is logged. If this automatic logging feature was to be extended into a more useful tool, it could be very beneficial for the Overture tool in general, since users could avoid writing their own logging mechanism for each model.

The customer was very impressed by the extensive tests which had been carried out on the model, and the log files from the test proved to be a great communication tool between the customer and the systems engineers in charge of the project. The test results also increased the confidence in the proposed solution for the development team.

For a system the size and complexity of the one presented here, it is very difficult (if not impossible) to analyse the many combinations of system state and AS input by hand. In addition the manual approach is very error prone, which could result in agreeing on erroneous behaviour and not discovering critical design flaws in the protocol. The test suite composed for this project does not ensure complete coverage of the state-space of the system, but provided a simple framework enabling extension of other scenarios to analyse some newly discovered corner-case. This ensured a short duration of the iterative cycles internally in the company when new corner cases had to be tested.

For the systems engineers leading the project, this was their first experience with formal methods, and in using executable models to specify functional requirements in general:

> *"The possibility to run numerous scenarios to analyse different combinations of ECAP system state and AS input was invaluable, and the rapid feedback from the model designer was very useful due to the time constraints of the project. We are extremely happy with the results obtained from this case study, which helped us in reaching an agreement with the customer within a very limited period of time."*

The models of ECAP, AS and the protocol were developed by a single person over a period of just two months including knowledge gathering of the systems involved. This was only possible due to the fact that a lot of details of the real system was abstracted away, and only the main functionality of the systems was included in the model. The different subsystems are connected by a military standard communication bus, which could have been modelled in detail to test package collision etc. In addition, several subsystem commands to enable and disable various subsystems were omitted. This is indeed one of the main advantages of using system modelling in the early phases of system development; describe the parts of the system of interest in detail and abstract away from any unneeded details. For example, low level implementation details of the desired logic of various drivers is not needed to understand the overall functionality of the system – hence abstracting away from such details helps creating more readable models giving a better system overview.

6 Concluding Remarks

Specifying requirements of any software system can be a very arduous task, and requires good communication between the customer who is the domain expert and the software engineers developing the system. For complex systems this is even more true,

since even small misunderstandings or misinterpretations can lead to erroneous design. An industrial case of a self-defense system used on-board fighter aircraft has been presented, where the use of an existing protocol used between the main application of the system and an intelligent sensor attached to the system had to be changed. An executable model was developed using the formal specification language VDM, and a voluminous test suite of scenario-based tests were used to exercise the model. The resulting output was analysed, and used to reach an agreement with the customer concerning the functional requirements of the use of the protocol.

The use of a formal model and lightweight formal methods analysis principles such as the scenario based tests and manual inspection of the generated proof obligations proved to be very valuable for the project. A lot of insight was gained in the functionality of the system in general and specifically of the new interpretation of the messages passed between ECAP and the AS subsystem which was invaluable in reaching an agreement with the customer. The lightweight approach to formal analysis of the model proved to be sufficient for the problem at hand: the development team had to be confident with the proposed solution, and the customer needed to be convinced that the correct solution had been identified. It is possible to translate VDM proof obligations automatically [25,26] to the theorem prover HOL [9] where these can be formally verified. Compared to this more strict formal approach a lot of time was saved using the more lightweight approach with manual proof obligation inspection and scenario based tests.

A new feature of the Overture tool is combinatorial test of models [16], which automatically generates and executes a large collection of test cases derived from trace definitions which are templates added to a VDM specification. These trace definitions are defined as regular expressions describing possible sequences of operation calls, and all possible combinations of these operation calls are expanded and executed automatically by the tool. This is a great tool to detect run-time errors caused by forgotten pre- or postconditions or broken invariants. Future plans includes making use of this test feature to better cover the large state space of the ECAP model.

All parameters of the system were modelled statically in this case study. In the real system, an XML-based file includes all the system parameters that are used to configure the system. Future work on this case includes parsing the data from this XML file into the model, so it can be dynamically configured just like the real system. This is possible due to recent extensions to the Overture IDE [17] enabling the model developer to access functionality defined in an external jar file. Using this extension it is possible to use a Java XML parser to parse the content of the configuration file, and access the data from the VDM model.

Only a subset of the entire system was included in the model, which was developed in parallel with the team of engineers developing the system, and was as such not an integrated part of the development process. For this to scale to larger modelling tasks of complete systems, a higher degree of integration into the daily development is needed. SCRUM was chosen based on its very agile approach to software development, so for formal modelling to fit into this, it needs to be somewhat agile as well. Even though they could seem like two orthogonal approaches to software development, various discussions concerning the combination of formal and agile methods have already

taken place [21,1]. Further work is planned on a methodology integrating formal modelling into agile development processes, and thereby combining the best of two different worlds.

Acknowledgements The author would like to thank Nick Battle and Peter Gorm Larsen for valuable input on the work presented here, which is partly funded by "The Danish Agency for Science, Technology and Innovation". In addition, Terma A/S deserves thanks for believing in the project in these trying times. Finally, the author thanks the development team for their great interest in the project and for good collaboration.

References

1. Black, S., Boca, P.P., Bowen, J.P., Gorman, J., Hinchey, M.: Formal versus agile: Survival of the fittest? IEEE Computer 42(9), 37–45 (September 2009)
2. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The jml and junit way. In: ECOOP 2002, volume 2374 of LNCS. pp. 231–255. Springer (2002)
3. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc
4. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
5. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://www.vdmbook.com`
6. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices 43(2), 3–11 (February 2008)
7. Fröhlich, B.: Towards Executability of Implicit Definitions. Ph.D. thesis, TU Graz, Institute of Software Technology (September 1998)
8. Fuchs, N.E.: Specifications Are (Preferably) Executable. Tech. Rep. 91.10, Institut für Informatik, Universität Zürich (Juli 1991)
9. Gordon, M.: HOL: A Proof Generating System for Higher-Order Logic. In: Birtwistle, G., Subrahmanyam, P.A. (eds.) VLSI Specification, Verification, and Synthesis. Kluwer Academic Publishers (1987)
10. Hayes, I., Jones, C.: Specifications are not (Necessarily) Executable. Software Engineering Journal pp. 330–338 (November 1989), `http://www.cs.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-89-12-1.html`
11. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)
12. Kramer, J.: Is Abstraction the Key to Computing? Communications of the ACM 50(4), 37–42 (2007)
13. Kurita, T., Nakatsugawa, Y.: The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
14. Kurita, T., Chiba, M., Nakatsugawa, Y.: Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods. pp. 425–429. Lecture Notes in Computer Science, Springer-Verlag (May 2008)

15. Larsen, P.G., Fitzgerald, J.: Recent Industrial Applications of VDM in Japan. In: Boca, B., Larsen (eds.) FACS 2007 Christmas Workshop: Formal Methods in Industry. BCS, eWIC (December 2007)
16. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010 (September 2010)
17. Larsen, P.G., Lausdahl, K., Ribeiro, A., Wolff, S., Battle, N.: Overture vdm-10 tool support: User guide. Tech. Rep. TR-2010-02, The Overture Initiative, www.overturetool.org (May 2010)
18. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In: Bicarregui, J., Fitzgerald, J. (eds.) Proceedings of the Second VDM Workshop (September 2000), Available at www.vdmportal.org
19. OMG: Omg unified modeling language (omg uml) infrastructure, v2.2. Tech. rep., http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/ (2009), oMG Available Specification without Change Bars, formal/2009-02-04
20. Overture-Core-Team: Overture Web site. http://www.overturetool.org (2007)
21. Peter Gorm Larsen, Sune Wolff, N.B.J.F., Pierce, K.: Development process of distributed embedded systems using vdm. Tech. Rep. TR-2010-02, The Overture Open Source Initiative (April 2010)
22. Ribeiro, A.: An Extended Proof Obligation Generator for VDM++/OML. Master's thesis, Minho University with exchange to Engineering College of Arhus (July 2008)
23. Schwaber, K.: Agile Project Management with Scrum. Prentice Hall (2004), iSBN: 073561993X
24. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085 (2006)
25. Vermolen, S.: Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department (August 2007)
26. Vermolen, S., Hooman, J., Larsen, P.G.: Automating Consistency Proofs of VDM++ Models using HOL. In: Proceedings of the 25th Symposium On Applied Computing (SAC 2010). ACM, Sierre, Switzerland (March 2010)
27. Wolff, S., Larsen, P.G., Noergaard, T.: Development Process for Multi-Disciplinary Embedded Control Systems. In: EuroSim 2010. EuroSim (September 2010)
28. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys 41(4), 1–36 (October 2009)

Sune Wolff:
Using Executable VDM++ Models in an Industrial Application - Self-defense System for Fighter Aircraft, 2012

Department of Engineering
Aarhus University
Edison, Finlandsgade 22
8200 Aarhus N
Denmark

Tel.: +45 87152101