International Journal of Mechanical, Industrial and Aerospace Sciences
ISSN: 2517-9950
Vol:2, No:2, 2008

Concerns Regarding the Adoption of the Model Driven Architecture in the Development of Safety Critical Avionics Applications

Benjamin Gorry

Abstract**—**Safety Critical hard Real-Time Systems are ever present in the avionics industry. The Model Driven Architecture (MDA) offers different levels of model abstraction and generation. This paper discusses our concerns relating to model development and generation when using the MDA approach in the avionics industry. These concerns are based on our experience when looking into adopting the MDA as part of avionics systems development. We place emphasis on transformations between model types and discuss possible benefits of adopting an MDA approach as part of the software development life cycle.

Keywords**—**Model Driven Architecture, Real-Time Avionics Applications.

I. Introduction

IN recent years there has been an increase in the number of companies adopting the Object Management Groups' (OMG) Model Driven Architecture (MDA) approach [1]. The model driven approach to system development facilitates better understanding of system requirements capture, design, construction, and generation. Transformations are used to convert one model type into another model type. In [1], when discussing transformations the following was stated:

"There are many ways in which such a transformation may be done. However it is done, it produces, from a platform independent model, a model specific to a particular platform."

Thus there is no generic approach to defining transitions as part of the MDA.

Hard real-time systems are ever present in the safety-critical domain of avionics applications [2]. A hard real-time system is where failure to meet a specified deadline can potentially lead to catastrophic consequences [3]. A hard real time system must be computationally correct and exhibit an acceptable degree of timeliness. Since the applications being developed will spend a large section of their active time in the air, failure of these applications is unacceptable – safety is of the highest priority. This idea of safety leads us to the concept of dependability. Dependability is described in [4] as:

"the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers."

Dependability relates to fault tolerance where concepts of what an error is and how faults are managed present themselves. This is further complicated in avionics applications since the majority of real-time systems are embedded. In embedded real-time systems the hardware and software are treated as a self-contained product, therefore the communications medium and architecture must also be taken into account during system development. In order to utilize shared resources, many of these systems take a distributed form [5]. The use of shared resources can create resource contention problems, but at the same time, the reduction in the quantity of resources required may also reduce the development costs of a system.

Modular approaches to software development have spread from the mainstream software engineering community to the development of avionics applications. This modularity provides two benefits:

- Components can be developed in isolation with the provision of well defined interfaces.
- Components, once verified for correctness, can be documented for future use and maintainability.

This modularity also spawns the possibility of utilizing mainstream approaches to software development such as the OMGs Unified Modeling Language (UML) [6]. This allows the avionics industry to begin to look into adopting a more commercial approach to software development. However, as attractive as these approaches are, avionics applications must conform to stringent development standards, as outlined in [7] and [8]. These outline a number of safety, development, and documentation requirements. Each of these must be shown to be fulfilled. To show that these requirements have been met formal approaches to verification, such as Model Checking [9], have been adopted. Problems associated with adopting the MDA approach when developing safety critical software have been discussed in [10].

This paper discusses issues relating to the adoption of an MDA approach as part of the development life cycle of avionics applications. Section 2 provides some background to

B. Gorry is with BAE Systems Rapid Engineering, Military Air Solutions, Warton Aerodrome, Preston, Lancashire, England.

the MDA and transformations. Section 3 presents an architectural view of a proposed MDA approach for use in the safety critical avionics domain. Section 4 discusses the transformations between different model types. Section 5 provides a focused discussion on the possible benefits of adopting the approach outlined in sections 3 and 4. Finally, section 6 lists the conclusions of our research.

II. MDA and Transformations

The Model Driven Architecture (MDA) was covered briefly in Section 1 of this paper. This section takes a more detailed look at the MDA with particular focus placed on the types of models and transformations between models types.

A. A Separation of Concerns

When using the MDA we discuss terms such as modularity, differing model types, and transformations between model types. By separating the details of a system development into different models we can focus our attention on a particular point of the system development process. Dijkstra describes this as a *"separation of concerns"* [11]:

> *"This is what I mean by "focusing one's attention upon a certain aspect"; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent they are irrelevant for the current topic."*

In the MDA, the concerns which are separated are different model types. When we have a lucid understanding of the required model types, we can then begin to establish a number of transformations between them.

By using the MDA several benefits are provided, such as:

- *Productivity* – this is improved since a developer can shift their focus from writing source code to developing an abstract model. This allows them to focus on solving the problem rather than being weighted down with implementation details.
- *Portability* – platform specific details can be isolated by means of using platform independent models.
- *Maintenance and Documentation* – since a model is an abstract representation of the source code, the model, to a certain degree, fulfils the function of high-level documentation

B. MDA Model Types

In the MDA specification [1], several model types are outlined. The most commonly referred to model types are PIMs (Platform Independent Models) and PSMs (Platform Specific Models). A PIM represents the details of a system without relating it to a specific platform (i.e. programming language or hardware construct). The PSM views the system from the specific point of a chosen platform.

Fig. 1 Illustration of PIM to PSM Transformation

To move between a PIM and a PSM, the MDA specification [1] discusses the concept of model transformations. Fig. 1, adapted and taken from [1], illustrates this in its basic form. In addition to the transformation of the platform independent system details from the PIM to the PSM, it may be necessary to augment the PSM with platform specific data. The PIM can be thought of as an abstraction of the desired system; this allows multiple PSMs to be derived from this abstracted PIM. What would be required is a number of different transformations and possibly additional information relating to the specific problem domain. This leads us to a discussion of the different transformation types which are available.

C. A Corpus of Model Transformations

The OMG "Request for Proposal" [12] for the MOF (Meta Object Facility) [13] version 2.0. outlined a request to support QVT (Queries / Views / Transformations) for models. The MOF is an industry standard, developed by the OMG, to facilitate the exporting of models from one application to another, the importing of models, and the translating of models into different formats. In QVT, *queries* are expressions which are evaluated over an entire model, *views* are models which are completely derived from other models, and *transformations* are used to generate target models from source models [14]. In [15], the OMG outlines that a transformation consists of a number of "mappings". M*appings* may be:

- Vertical (changes are made to the source model and disseminated to the target model),
- Horizontal (these describe relationships between different views),
- Unidirectional, or
- Bidirectional (synchronized).

There are differing approaches to declaring mappings, each of which agree that a mapping must consist of a domain and a range. In [14] two approaches to declaring mappings are discussed; *imperative* and *declarative* mappings. *Declarative* approaches represent relationships between source and target components. These relationships often take the form of a fixed set of inference rules or functions. *Imperative* approaches explicitly state a sequence of steps which must be followed to produce the required result. The most commonly used is the declarative approach.

$$(1) \rightarrow (a)$$
$$(2) \rightarrow (b)$$
$$(3) \rightarrow (c)$$

(3) → (d)
(4) → (e)
(*) → (z)

Fig. 2 Declarative Transformations

Fig. 2 provides an example of six declarative mappings. These mappings take the form of an integer value at the source, this maps to a character value at the destination point of the mapping. Each mapping is vertical and unidirectional. The first two mappings in Fig. 2, along with the fifth mapping, are straightforward integer to character mappings. However, as we can observe in the mappings on lines 3 and 4 of Fig. 2, both mappings have the same domain value. In this case we cannot always guarantee what the range value will be, it could be either of the characters ‘c’ or ‘d’. The sixth and final mapping consists of what is known as a “wildcard” value (this will recognize any of the five possible domain values as being valid), hence the wildcard value is valid at any time a valid integer value is present in the domain; making rule selection non-deterministic and increasing the possibility of an undesirable range value occurring. This simple example illustrates how important it is to define deterministic (dependable) mappings.

If we then make these mappings bidirectional, as shown in Fig. 3, this further complicates matters. In Fig. 3, from range values ‘c’ and ‘d’ we map back to domain value ‘3’. If we then map from domain value ‘3’ we are not guaranteed to map back to our original value of ‘c’ or ‘d’. We could repeat

(1) ↔ (a)
(2) ↔ (b)
(3) ↔ (c)
(3) ↔ (d)
(4) ↔ (e)
(*) ↔ (z)

Fig. 3 Bidirectional Transformations

this process continually in both directions without us ever being able to map to our desired range value. If mappings between models are bidirectional we can say that this exhibits a degree of synchronization between the models. The greatest care must be taken when defining even the most basic of mappings if true synchronization of models is to be achieved.

III. Proposed Approach

To adopt an MDA-based approach as part of the development cycle of avionics applications the transformations between model types must be well-defined. Tools such as FeaVer (Feature Verification) [16] and frameworks such as PARTES (Performance Analysis of Real-Time Embedded Systems) [17] have been developed. These facilitate the transformation of program source code into formal models for analysis. In tools like these, the transformations which are used are based on abstraction decisions made by the tool developers. We must then ask ourselves two questions:

1. How can we be certain that the abstraction decisions made by the tool developers are sound?
2. In keeping with the MDA approach, how can we define transitions in a tractable fashion so that they can be easy altered, if required, at a future point in time?

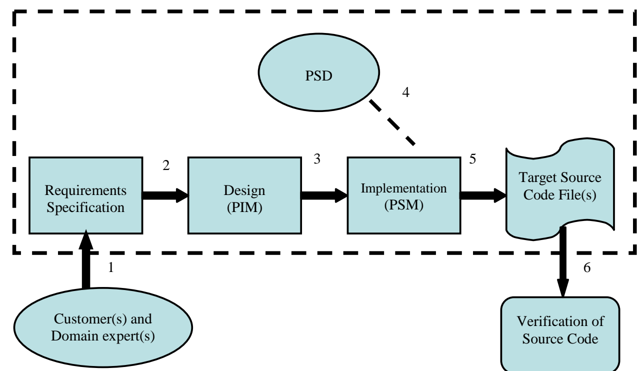

Fig. 4 An MDA Approach for the Development of Avionics Applications

Attempts have been made to address these points. Mapping languages such as XMap [18] exist. XMap is designed to support unidirectional mappings which are described in terms of patterns which describe what a mapping does. Patterns represent a link between the source and target models and are declarative. However, since the form that these mappings take is structured round the abstraction decisions made by the developers we must remember that they may not fulfill our requirements. It may be the case that bidirectional mappings would be more appropriate in some examples of our models; XMap does not facilitate the use of bidirectional mappings. Also, if unidirectional mappings were acceptable at the present time, it may be the case that we will require bidirectional mappings after *n* revisions of our software – hence the mappings produced may not be suitably tractable.

We focus on the issue of transformations because in the scope of developing an avionics application several different types of transformations exist. Over a project lifecycle of twenty to thirty years, if the transformations between each of these stages are not clearly defined this could have huge cost implications for the project. A diagram representing a simplified MDA approach to avionics systems development is displayed in Fig. 4. This diagram illustrates the approach which is proposed for use when developing avionics applications.

The transformations listed in Fig. 4 take five different forms; these are represented by the black arrows which are numbered and are displayed as being unidirectional to show the flow of progress through the various development stages.

However, these transformations may also be bidirectional. These will be discussed in detail in the section that follows. The dashed-line box bounds the internal model transformations; these are classified as this since the system developer should not be required to know anything about them. Human interaction with the approach is restricted to those elements that lie outside the bounding box.

The issues with MDA transformations are discussed in [19]. Paper [19] discusses how the Object Management Groups specification for the Common Warehouse Metamodel (CWM) specification [20] describes transformations. However, as stated in [19], the CWM does not describe a methodology for implementing transformations; it provides a model for describing the existence of mappings. Other approaches to model transformation such as those found in graph theory are discussed in [19]. This approach has been taken in tools such as [17] and involves a set of rules which describe mappings from source to target models where either or both models contain graph-like structures. This approach is common in tools which employ Petri net [21] approaches.

In Fig. 4, the seven boxed areas relate to general areas of systems development; an overview of these areas is now provided. Firstly, depending on the aircraft being developed, system requirements are captured from the target customer(s) with assistance from experts in the development of similar products. From these requirements a system design is formulated. The design is then transformed into an implementation of the target system. This also includes Platform Specific Data (PSD) elements (represented as linking into the implementation stage via a dashed line in Fig. 4). These may include new hardware features which are to be incorporated as part of the system, or specific features for different evolutionary developments of the basic design - these elements consist of actual application data which is required as part of the implementation. From the implementation stage, the target program source code is automatically generated. Finally, this source code is verified using formal approaches to program verification such as testing. We now discuss the various transformations which take place between these stages.

IV. Transformations between Stages

In Fig. 4, six transformations between the stages of a proposed MDA approach are listed; these are now discussed. The first transformation involves capturing the requirements of the target system from the customer(s) with the assistance of domain experts. This stage generally involves formulating a natural-language textual description of the requirements that must be fulfilled. These requirements can then be specified formally in a notation such as, for example, Z [22] or the Vienna Development Method (VDM) [23]. At this stage of the process the customer(s) have the opportunity to state any particular features they wish the system to contain. This transformation leads us into the internal section of the approach. In theory, it should not be necessary for us to discuss the internal working of the MDA transformations; but for illustrative purposes this is a useful exercise.

The second stage of transformations involves moving between the requirements specification to a design representation. A design generally takes a graphical form, such as a UML [6] diagram, which allows the target system to be viewed from a number of different angles. If the requirements are well-defined in a formal representation, such as Z, a set of mapping rules (transformations) can be defined. This is where the first area of concern with an MDA approach arises. It is understandable that we have to rely on human-based decisions to formulate a requirements specification. At the requirements stage we are guided by notations such as Z, but when we move from requirements to design we rely on abstraction decisions made by the engineer who is responsible for developing the transformation rules.

It would be expected that the transformations are unidirectional, however it may be beneficial to utilize bidirectional transformations as we may wish to change the design at the graphical, i.e. UML, stage and then witness the corresponding changes in the requirements specification. Before we discuss the remaining four transformations we shall address the issue of *abstraction*.

Abstraction is used as a means of providing scientific descriptions of the world [24]. A programming language can be thought of as an abstraction that has been developed to ignore details of specific machine instructions and architectures. Abstractions are limited by *domain*, *grain*, and *level* [25]. Domain refers to the statements and the types which are supported (in his case the types of mappings which are supported). When a programming language is developed it has a certain grain (or structure) which is core to the way in which it should be used. When using the language it is beneficial to use this grain by following the desired style, this is how to achieve maximum usage of the language. In the case of model transformations there may be a structure to developing these transformations. This grain will have been decided upon by the developer of the mapping structure. Therefore we have to trust that the developer has made appropriate abstraction decisions; this leads us to level. The level of abstraction which is employed must facilitate the development of lucid and tractable transformations. In avionics applications the size of the PIM and PSM will be considerable and may be extremely complex in certain areas. To assist in understanding the relationships between these models we should employ transformation languages which allow us to work at a design (and possibly flexible) level of abstraction for differing applications.

Transformation three involves moving from a PIM representation to a PSM representation which relates directly to the target hardware upon which the system will be deployed. Stage 4 inserts the PSD into the implementation PSM. Again, mapping from the PIM to the PSM raises issues of abstraction. At the end of the third and fourth transformations we should have in our possession a model which takes into account hardware components and software

interface points within these components.

The fifth transformation involves mapping the PSM model into an appropriate source code representation. For avionics applications, a typical language of choice is the high level programming language SPARK Ada [26]. This process of mapping from a PSM to program source code can be thought of as auto-code generation. Like the previous transformations which we have discussed, this also suffers from the problem of showing that the transformations perform as desired. If we do not develop an appropriate set of transformations we will generate incorrect code. In avionics systems we often deploy complex files containing a large number of lines of source code. Even if our transformations provide us with correct code, they may cause us to develop inefficient code which does not provide an optimum representation of the system software. This has implications on areas such as:

- Shared resources,
- Allocation of redundant components, and the deployment of an appropriate fault-tolerance strategy.
- The development of efficient code should be a central factor when considering which transformation approach to adopt when generating system source code.

The final transformation, number six, leads us out of the bounded box. This involves us verifying the source code for correctness. This is currently the most well defined transformation of Fig. 4. The SPARK Approach [26] (SPARK is a safety-critical subset of the Ada language) advocates design by construction through a series of program annotations.

A series of verification tools are available to analyze the Ada source code. In addition to this approach, traditional real-time source code testing approaches can be employed. These specify different levels of program testing and build in facilities for features such as stress and regression testing. This discussion of the seven transformations listed in Fig. 4 leads us to other concerns regarding model transformations.

A. Further Issues to Consider

When discussing transformations the issue of abstraction was frequently raised. This is a common problem and it has been given attention in fields such as model checking [9]. Since abstraction is regarded as being a common problem in modeling, guided approaches to model abstraction and the definition of model transformations have been developed. An example of this is PARTES [17]. Even if we have followed a guided abstraction approach and we are satisfied that the transformations act as required, how can we ensure that they are correct? From the point of view of the avionics industry, a structured argument which demonstrates that the model transformations which have been employed are both syntactically and semantically correct is required. A possible mechanism to ensure correctness would be to use Hoare-style [27] assertions which take the form of *pre* and *post* conditions. For example, in:

Pre {T} Post

where *Pre* is a precondition which must be met for transformation *T* to be selected and *Post* is the result of transformation *T*, we can say that if assertion *Pre* is true before transformation *T* takes place and assertion *Post* is true after transformation *T* takes place then the transformation is valid with respect to the specification *Pre {T} Post*, can't we?

For a simple unidirectional transformation this is acceptable. However, as outlined in [28], some transformations may contain internal transformations which are required to aid construction of intermediate representations (also known as rule organization [28]) required for use by other transformations. Also, it may be the case that transformations may be parameterized – this could increase the flexibility of the transformation approach by increasing the re-use factor of a transformation from a *1* to an *n-value* parameter range.

If transformation rules contained sub-rules, we may find that rules begin to grow in complexity as the levels and breadth of rule hierarchy's increase. In this situation we would have to place constraints on which rules can be coupled with other rules and in which order. If we did this, we could then use Hoare-style assertions to determine the overall correctness of a transformation rule by determining the correctness of the sub-rules which are used within it. In [28] the concept of "rule scheduling" to determine the order in which rules may be applied is discussed.

When we are satisfied that our transformations are correct, we must then look at what, if any, overhead is placed on the MDA process. Overheads may include:

- increasing the size of the transformation code, or
- increasing the time taken to map between PIM, PSM, and program source code.

For a large number of transformations, the time it may take to prove that these are correct may be considerable. Also, the idea of "rule scheduling" would require increased resources when executing transformations. This may act as a syntax check in relation to rule-ordering. However, as discussed previously, this would not provide a guarantee that the transformations which have been employed are correct.

In Fig. 4 the element of PSD was included. This data may refer to the hardware architecture which is being used. In modern day distributed architectures measures have been taken to ensure timeliness while employing fault-tolerant mechanisms to ensure that safety constraints are met. Four such architectures are:

- SAFEbus – used by Honeywell Aerospace,
- SPIDER – used by NASA,
- TTA - used by Honeywell Aerospace, and
- FlexRay – developed by a consortium including BMW.

An overview and comparison of these architectures is

provided in [29]. If an overhead is placed on ensuring that the transformation between PIM and PSM are correct, we must make sure that this overhead does not impact on the performance of the hardware architecture in any way. The cost of an overhead may defeat the purpose of using such an architecture.

In [30] Czarnecki and Helsen outline over twenty model-transformation approaches. In providing a list of these approaches we are given an opportunity to survey the approaches which are available with an aim to making a choice which will satisfy our requirements. Unfortunately, it appears that none of these approaches address the issues which have been discussed in this section. In [30], it is stated that:

"industrial-strength and mature model-to-model transformation systems are still not available, and the area of model transformation continues to be a subject of intense research."

This supports the findings of our assessment of current approaches. It appears that no current strategy for specifying model transformations, which addresses the issues discussed in section 4 of this paper, exists. The main reason for this is that the capabilities to formally verify the model transformations which are used do not exist. However, it is recognized that existing techniques (i.e. testing) exist to show whether or not the source code, which is produced from an MDA process, conforms to safety-critical standards.

When discussing the types of transformation rules which are available, [30] covers points which support some of our concerns; these relate to:

- rule reuse - rules should be structured generally to facilitate ease of reuse, and
- directionality – determining the direction which a transformation can be executed in.

The idea of rule re-use was identified as being a problem which will require further attention in [10]:

"it may require very careful engineering to produce transformations and meta-models which can actually be re-used in multiple contexts."

For directionality, we previously discussed unidirectional and bidirectional transformations. In [30] the concept of multidirectional transformations is suggested. To solve the issue of non-determinacy relating to the selection of mapping rules (as discussed in section 2 of this paper), [30] suggests *application conditions*. Application conditions are conditions that must be true before a rule will fire. application conditions are part of a strategy which is used. A strategy may be deterministic, nondeterministic, or interactive. In a deterministic strategy there will be a defined rule for each transformation scenario. In a non-deterministic strategy there may be more than one rule which is applicable in a given situation; and in an interactive strategy the engineer would be presented with a selection of rules. They would then make their choice of rule.

The idea of developing a strategy when it comes to the selection of transformation rules is a step in the right direction. Ideally, as discussed in section 2 of this paper, we would require a deterministic strategy. If this was in place for every possible model transformation in our system, we would be provided with a degree of assurance that the correct transformations were being selected at each stage of the MDA process. The final part of this section discusses current attempts which have been made to address some of the issues which have been raised.

B. Approaches to Proving Correctness of Transformations – Related Work

Several problem areas relating to transformations between model types have been discussed in the previous parts of this section. To address some of these issues, research has been carried out. One approach to showing correctness of transformations in the MDA involves using testing approaches [31]. In [31] it is discussed that that transformations should be expressed as operations within the OCL [32] (Object Constraint Language) – a language which can be used to enhance the precision of a model by associating assertions (constraints) with model elements. The OCL has been developed to assist in the removal of ambiguities from UML diagrams by using a notation which cannot alter the model, each statement simply returns a value as it is evaluated, and is easier to utilize than traditional formal approaches to specifying model constraints. In OCL, pre and post conditions take the form of assertions. As discussed in [30], pre-conditions specify constraints on the input parameters while post-conditions specify the effect of the transformation by linking the input and the output parameters. This sounds similar to the pre and post-condition concept raised in section 4.1 of this paper. Unfortunately, as for the reasons discussed in section 4.1 of this paper alongside [28], this does not provide suitable analysis of more complex transformations.

In [31] the idea of using *partition analysis* is raised. Partition analysis involves dividing the input domain of the program under test into several sub-domains than do not overlap. A unique test case is then chosen from each sub-domain. Partition analysis works under the assumption that if the program behaves correctly with one test case from each sub-domain, it should then be valid for all test cases [31]. Also, *"classes must be well chosen and the number of classes must be reasonable"*. Right away we must ask – how do we ensure that the choice of test case is correct? In [31] proposed methods of partitioning are discussed. However, for a large-scale avionics application we may be facing a possibility of creating tens of thousands of partitions; this breaks the requirement that the number of classes which are chosen should be "reasonable".

In [33], the verification of triple graph grammar transformations is discussed. Triple graph grammars are a specification technique which is used to specify model to source code transformation. A graph grammar rule is applied by substituting the domain of the rule with the target if the domain can be matched. In [33], verification of triple graph grammars is carried out using the theorem prover ISABELLE/HOL [34]. In this approach, abstraction decisions have been taken to transform representations of graph transformation rules into type definitions which are comprised of all the structured information from the graph transformation rules. The results of this can then be analyzed using ISABELLE/HOL. The problem with this approach relates to the abstraction decisions which are taken in order to transform the graph transformation rules into type definitions. Even if we assume that the abstractions are lucid; [33] indicates that a large number of lines of proof code are required to show that transformations are accurate. Since [33] does not discuss scalability of this approach; it is unclear whether such an approach could scale to a major avionics application.

Another approach is discussed in [35]. In [35] approaches to white-box testing are outlined. The authors of [35] have indicated that their approach partially overcomes the challenges of:

- generating test cases from model transformation specifications, and
- generating test oracles to determine the expected result of a test.

Firstly, constraints expressed in the OCL (as the work discussed in [31] outlined) can be used. These constraints can be used to construct interesting test cases; if a constraint is violated we are presented with a possible test case. In this situation, the test oracle is the execution environment. The execution environment checks the constraints after each transformation is applied. The form which the program constraints will take is not discussed. Also, as with [33], the scalability of this approach is not discussed. In section 4.1 of this paper, we discussed how transformations may contain sub-rules and these can contribute to the complexity of the transformation. In [35] this issue is raised with reference to ensuring the confluence of transitions which are dependent on one another. This involves ensuring that any conflicting transformation steps are captured and analyzed to decide whether or not a suitable approach can be identified. Again, the scalability of this approach is not discussed.

To bring this section to a close, the work presented in [36] is discussed. The approach in [36] is based on the approach taken in the development of application software. This is likened to the waterfall development life cycle and the author presents an approach which checks the syntactic correctness of basic rules. In this approach UML models are translated to CSP [37]. To ensure that the transformation is syntactically correct, the OCL constraints in the UML models are checked and the CSP part is checked against the syntax of CSP by replacing all non-terminal items in the CSP with terminal items. If both the UML and CSP models are syntactically correct, the transformation is said to be correct. This approach, like the others, has only been tried out on small scale systems. Also, it only relates to transformations between UML and CSP, there are no bidirectional or platform independent transformations.

V. Benefits of Proposed Approach

By being able to separate model types, and design our applications in a platform independent fashion the strength of MDA is evident. In large-scale avionics projects the cost of development reaches into a factor of billions of pounds. This cost increases if we have to continually construct models when the project evolves between stages. By embracing the concept of a PIM, two benefits are immediately spawned.

Firstly, by designing the application using a PIM approach, we are abstracting away from the concrete system view. Transitioning from our PIM design to the actual PSM would then take place via a number of model transformations. By developing models as PIM we can utilise notations such as the UML. Since the UML has been integrated as part of many University software engineering degrees; when employing new staff to work on projects they may already have some knowledge of how to use an approach. This would save on staff training and development costs. Since we can make a convincing argument for the use of notations such as the UML, and since the techniques used to develop PSM have a proven track record, this leaves us with having to provide a method of assuring that the transformations between these model types are reliable and will act deterministically.

The second benefit to adopting an MDA approach lies in technology change. By adopting a "separation of concerns" between our PIM and PSM we are beginning to plan ahead for future avionics applications or revisions to the current system. By adopting an approach such as that outlined in Figure 4, we are keeping the PIM, PSM, and PSD separate. This would allow us to alter the PSD and PSM at future stages of the project. This may be extremely useful if the desired target source code language changes or if one of the distributed architectures, such as those stated in section 4.1 of this paper, is adopted.

Outwith the standard benefits of adopting an MDA approach, the idea of transformation patterns is a possible concept which could emerge from a well-defined, reliable, and structured approach to model transformations. In object-oriented design there exists a range of design patterns [38] which have developed over the past few decades. Each of these patterns prescribes a solution for differing model architectures. In general, these patterns have been incorporated into notations such as the UML. These patterns, which are already used to develop MDA PIM, provide guidance on how to design systems from particular viewpoints. If we could build up a corpus of transformations we could utilize transformations of different types with reference to:

different design approaches (patterns), and other

transformations; with the aim of constructing well-defined hierarchical structures for the more complicated transformations. This would show that the time spent developing a reliable and tractable approach to model transformation would reap rewards in relation to what are now well-established approaches to object-oriented design.

VI. Conclusion

This paper has discussed the outstanding issues and concerns regarding the adoption of the Model Driven Architecture (MDA) as part of the development of avionics applications. This discussion has taken the form of referring to an outline of an avionics application which has raised a number of issues relating to proving the correctness of model transformations. Referring back to the two questions that we asked earlier:

1. How can we be certain that the abstraction decisions made by the tool developers are sound?
2. In keeping with the MDA approach, how can we define transitions in a tractable fashion so that they can be easy altered, if required, at a future point in time?

We can now say that, after researching current practical and theoretical attempts at answering these questions, that these questions remain unanswered at this point in time. Some of the work discussed in section 4.2 has made progress; however it will require further investigation and proof by application. If these questions could be answered, we are certain that this would be a significant step towards the adoption of the MDA as part of development of avionics applications. The benefits and potential cost savings during systems development are clear and may lead to reduced development costs while providing assistance to traditional approaches to systems development.

Acknowledgement

Thanks go to Kevin Dockerill and John Rowlands for their useful feedback and comments.

References

[1] Object Management Group. MDA Guide Version 1.0.1, Technical Guide, Object Management Group, June 2003.
[2] Cheng A. M. K. Real-Time Systems – Scheduling, Analysis, and Verification. John Wiley & Sons, Inc, 2002.
[3] Burns A., Wellings, A. Real-time Systems and Programming Languages. Pearson Education Limited, 2001.
[4] Laprie J-C, Dependability: Basic Concepts and Terminology. Springer-Verlag Wien New York, 1991.
[5] Coulouris G., Dollimore, J., Kindberg, T. Distributed Systems – Concepts And Design. Pearson Education Limited, 2001.
[6] Pooley R, Stevens P, Using UML – Software Engineering With Objects and Components. Addison-Wesley, 1999.
[7] RTCA-EUROCAE. Software Considerations In Airborne Systems and Equipment Certification. Do-178B/ED-12B. RTCA and EUROCAE, 1992.
[8] Ministry Of Defence. Safety Management Requirements for Defence Systems. DEF-STAN 00-56, Draft Issue 3, UK Ministry Of Defence, 2004.
[9] Clarke E. M., Grumberg O., Peled D. A., Model Checking. MIT Press, Cambridge, Massachusetts, 1999.
[10] Conmy P., Paige R.F., Challenges when using Model Driven Architecture in the development of Safety Critical Software. Proceedings of 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES), 2007.
[11] Dijkstra E. W., A Discipline of Programming. Prentice Hall Series In Automatic Computation, Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1976.
[12] Object Management Group, Request For Proposal: MOF 2.0 Query / Views / Transformations RFP. Object Management Group, 2002.
[13] Object Management Group, Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group, 2004.
[14] Gardener T., Griffin C., Koehler J., Hauser R., A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. OMG Document: ad/03-08-02.
[15] Object Management Group, Meta Object Facility 2.0 Query/View/Transformation Specification. 2005. http://www.omg.org/docs/ptc/05-11-01.pdf
[16] Holzmann G. J., Smith M.H., An Automated Verification Method for Distributed Systems Software Based on Model Extraction. IEEE Transactions On Software Engineering, 28(4):364-377, 2002.
[17] Gorry B., Ireland A., King P., PARTES: Performance Analysis of Real-Time Embedded Systems. Proceedings of 4th International Conference on the Quantitative Evaluation of Systems (QEST), pg 271- 272, 2007.
[18] Clark T., Evans A., Sammut P., Willans J, Applied Metamodelling: A Foundation for Language-Driven Development Version 0.1. www.xactium.com .
[19] Gerber A., Lawley M., Raymond K., Steel J., Wood A., Transformation: The Missing Link of MDA. Proceedings of the First International Conference on Graph Transformation (ICGT), 2002.
[20] CWM Partners, Common Warehouse Metamodel (CWM) Specification. OMG Documents: ad/01-02- {01,02,03}, February 2001.
[21] Petri C. A., Communications with Automata. Technical Report RADC-TR-65-377, New York, 1966.
[22] Spivey J. M., The Z notation: a reference manual. Prentice-Hall International Series In Computer Science, 1989.
[23] Jones C. B., Software Development: A Rigorous Approach. Prentice Hall International, 1980.
[24] Ben-Ari M., Principles of Concurrent and Distributed Programming. Prentice-Hall International, 1990.
[25] Dix A. J., Formal Methods for Interactive Systems. Academic Press, 1991.
[26] Barnes J., High Integrity Ada: The Spark Approach. Addison-Wesley Professional, 1997.
[27] Hoare C. A. R., An axiomatic basis for computer programming. Communications of the ACM, 1969.
[28] Czarnecki K., Helson S., Classification of Model Transformation Approaches. Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.
[29] Rushby J., A Comparison of Bus Architectures for Safety-Critical Embedded Systems. Technical Report, Computer Science Laboratory, SRI International, 2001.

[30] Czarnecki K., Helsen S., Feature-based survey of model transformation approaches. IBM Systems Journal, Volume 45, Number 3, 2006.
[31] Fleurey F., Steel J., Baudry B., Validation in Model-Driven Engineering: Testing Model Transformations. Proceedings of the First International Workshop on Model, Design and Validation, pg 29-40, 2004.
[32] OMG, Object Constraint Language (OCL), OMG Available Specification, Version 2.0. 2006. http://www.omg.org/docs/formal/06-05-01.pdf
[33] Giese H. et al., Towards Verified Model Transformations. Proceedings 9th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 2006.
[34] Nipkow T., Paulson L. C., Wenzel M., Isabelle/HOL : A Proof Assistant for Higher-order Logic. Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2002.
[35] Kuster J. M., Abd-El-Razik M., Validation of Model Transformations – First Experiences using a White Box Approach. Proceeding of Model Driven Engineering Languages and Systems (MoDELS), pg 193-204, 2006.
[36] Kuster J. M., Systematic Validation of Model Transformations. Proceedings of the 3rd UML Workshop in Software Model Engineering (WiSME), 2004.
[37] Hoare C. A. R., Communicating Sequential Processes. Prentice-Hall International, 1985.
[38] Gamma E. et al, Design Patterns : Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.