8^{th} ICCRTS
17–19 June 2003, Washington DC
Track 7: Information Superiority/ Information Operations

Security Modelling for C2IS in UML/OCL

Robert Charpentier
(418) 844-4000 x 4371
robert.charpentier@drdc-rddc.gc.ca

Martin Salois
martin.salois@drdc-rddc.gc.ca

DRDC Valcartier
2459 Pie-XI Blvd. North
Val-Bélair, QC, Canada
G3J 1X5
(418) 844-4538 (fax)

Security Modelling for C2IS in UML/OCL

Robert Charpentier
robert.charpentier@drdc-rddc.gc.ca

Martin Salois
martin.salois@drdc-rddc.gc.ca

DRDC Valcartier
2459 Pie-XI Blvd. North
Val-Bélair, QC, Canada
G3J 1X5

Keywords: Software certification, security modelling, high-confidence software

Abstract

As a C2IS centre of expertise, DRDC Valcartier devotes part of its R&D program to the security and reliability of military information systems. The MaliCOTS project (Malicious Code Detection in COTS Software; 1997-2001) demonstrated that the risk associated with critical software could be managed in large part by:

i Execution monitoring (e.g. surveillance of the programs and users' behaviour),

ii Static verification of code prior to execution (e.g. control flow safety),

iii Certification at compile time (e.g. buffer overflows and type safety).

One of the main conclusions of the MaliCOTS project was that it is mandatory to manage software risks early in the development cycle. This provides for better management at a lower cost.

Since modelling aspects were not studied in that project, a new one called Secure OCL expression (SOCLe) was initiated to study the certification of Command and Control Information System at the design phase. The primary objective is to manage as much of the risk as possible early in the software development cycle, thus avoiding subsequent software revisions that are usually expensive and counter-productive.

In this article, the relevant research from the past five years is summarized and there is also a description of the secure development cycle that is currently being prototyped.

List of Acronyms

CASE	Computer-Aided Software Engineering	**OCL**	Object Constraint Language
C2IS	Command and Control Information System	**OMG**	Object Management Group
COTS	Commercial-off-the-shelf	**OS**	Operating System
DBC	Design-by-contract	**SOCLe**	Secure OCL expression
DRDC	Defence Research and Development Canada	**UML**	Unified Modelling Language

1 Why Certify C2IS Software

Software quality standards have improved greatly over the past ten years. They now appear to be adequate for most domestic and commercial uses. However, for critical systems such as flight-control and military systems, reliability and security requirements are much more difficult to attain because of the intrinsic complexity of combining independently-designed components. Indeed, modern Command and Control Information System (C2IS) are typically assembled from software components and sub-systems that are:

1. Commercially available and/or
2. Government-owned and/or
3. Obtained via collateral exchange programs and/or
4. Open-Source.

The quality of such programs varies considerably, as does their compatibility, leaving military systems wide-open to reliability problems, security risks, and maintenance difficulties.

Many mass-market devices, not necessarily safety-critical in nature, are now carefully debugged so as to avoid the dramatic impact of a mistake that may be discovered only after the product has been distributed. The Intel Pentium floating-point division unit fiasco is a famous example of an error that caused major financial losses (estimated at $500 million US [9].)

In the software industry, most editors are now acknowledging their responsibilities in terms of software quality. However, there is still a considerable amount of work to be done! Flaws are discovered every day and end-users are getting more and more upset by the burden of having to apply multiple software updates and repetitive patches in order to maintain their applications at an acceptable vulnerability level. According to Gartner [8], with the increase in electronic commerce, the financial impact of poor system design and poor security practices will continue to grow: "*Through 2004, the economic value represented by cybercrimes will increase by 2 to 3 orders of magnitude — i.e. 1000% to 10, 000%*"

Clearly, the motivation to have better quality software is very strong and should continue to increase for both the military and civilian communities for the foreseeable future. To achieve this increase in security and reliability, many aspects of system development need to be improved, including development methodologies, programming languages (e.g. Java), and various protection technologies (e.g.. firewalls, intrusion detection systems, etc.)

As a centre of expertise in C2IS, Defence Research and Development Canada (DRDC) Valcartier devotes part of its R&D program to the security and reliability of military information systems. Emphasis is placed on the Centre's spectrum of responsibilities, which include software certification techniques and rigorous ways of specifying and managing security requirements for C2IS.

The first part of this article presents an overview of these certification techniques, along with a brief summary of the experimentation that was carried out regarding these concepts. The goal was to highlight their strengths and weaknesses in the context of the detection of malicious code in Commercial-off-the-shelf (COTS) software (the MaliCOTS project) .

The second part of this paper focusses on current research that aims at starting the security process at the design phase in order to manage risk throughout the development chain. In the

SOCLe project, Unified Modelling Language (UML) diagrams are used to model a C2IS and Object Constraint Language (OCL) is used as a property specification language. The constraints are model-checked for conformity against a more universal security/reliability policy.

2 How is Software Certified

Traditionally, the risks associated with complex software systems have been mitigated by static code analysis and execution monitoring of software packages prior to integration. In order to assess the effectiveness of these well-known techniques for the detection of malicious code in the context of C2IS software, the four-year MaliCOTS project was initiated in 1997. It was discovered that each technique has its own strengths and weaknesses and that they can be organized in complementary manner. Three techniques were investigated and the following is a summary of the conclusions.

2.1 Technique #1: Execution Monitoring

Execution monitoring examines the behaviour of the program while it is running. In the prototype, called DaMon [6], this was achieved by inserting specialized drivers into the Operating System (OS). These drivers control access to critical resources such as files, communications ports, the Registry, and process creation/destruction.

Dynamic monitoring is a pragmatic approach that offers several short-term benefits, since it uses all the information available during execution, including the user's inputs and the network transactions. However, although many variants are available, as mentioned in the state of the art report [11], exhaustive and formal certification is rarely achievable. The main limitation is associated with the difficulty of identifying a complex behaviour emerging from a long trail of small OS transactions. Also, monitoring is reactive in nature and a malicious program could deliver its payload before it could be stopped.

Tests have shown that dynamic monitoring must be focussed on specific tasks for it to be usable in practice. Not every task can be efficiently handled by execution surveillance. For example, identifying dead code in software systems is not dynamic analysis' strongest point. On the other hand, surveillance of user behaviour, transactions with the network and monitoring of concurrent processes appear to be natural application domains (if not the only way of managing such risk!)

2.2 Technique #2: Static Analysis of Code Prior to Execution

Static analysis of code is common in the world of program optimization and software analysis. It consists in examining the code, perhaps in some abstract representation, without actually running it. When the source code is not available (as is the case with much of the COTS software), the certification must be done on the binary. This is extremely difficult, if not impossible. Interpreting control flow graphs requires highly-skilled experts who will often encounter undecidable mathematical conditions. Copyright issues might also preclude the disassembling or decompiling of executable code for analysis in some abstract representation.

Among the strengths of this technique, identification of dead code and hidden functionalities (e.g. the MS Excel'97 flight simulator that has nothing to do with a spreadsheet program) has been demonstrated in the past. More generally, static analysis has the ability to detect inappropriate logic by exploring all possible execution paths in a generic manner, as demonstrated by the SamCOTS static analyzer prototype [1]. A large number of commercial and research products (market survey [12]) have also confirmed the great interest in program visualization and code understanding.

Static analysis and execution monitoring are complementary techniques. They can be cascaded to maximize their efficiency. However, both techniques require highly-skilled experts and a considerable amount of time in order to perform a good analysis. In fact, it is often criticized as being too time-consuming for practical purposes. In addition, such a manual certification is also prone to errors as in any other human-driven activity.

In the context of the MaliCOTS project, it was observed that traditional techniques work well when the code to be analyzed is rather small, for example, a virus or some embedded software for microcontrollers. They are not magical solutions that will provide fast and highly-trustable certification of large applications such as those used in C2IS. They are difficult to maintain over the life cycle of large software packages, especially if there is no access to the source code, since periodic upgrades must be re-certified. Still, static analysis and execution monitoring are commonly used since they offer a realistic short-term solution to the management of the risk associated with untrusted software. These general conclusions provided the principal motivation to explore more automated techniques such as the certifying compiler, which is presented below.

2.3 Technique #3: Certification at Compile Time: An Emerging Technique for Automated Certification

To accelerate and formalize the certification process, the concept of an intelligent compiler has already shown great potential. The concept is described in detail in [5] and is illustrated in Figure 1 (from [4].) Basically, a security policy is formulated and then compared with the code being developed through a verifier that performs a rigorous code conformity check.

R&D on the detection of malicious code confirms that certifying compilers have the potential to structure and normalize the integration of trusted components into critical systems. The principal advantages of using an intelligent compiler include:

a Certification can be detailed and exhaustive even for very large software packages;

b Execution time will not be increased by the security mechanism (unlike the case of wrappers and interceptors);

c Deployment can be rapid because no (or very little) manual certification is needed (unlike static and dynamic analyses of pre-packaged software);

d With appropriate engineering, this approach can also facilitate other kinds of certification, including interoperability compliance, reuse policy and maintainability/reliability specifications;

e Editors can deliver a trusted component without revealing the intellectual property inherent in the code; and

f The concept of annotated components can be adapted to manage the risks associated with mobile code (e.g. self descriptive applets).

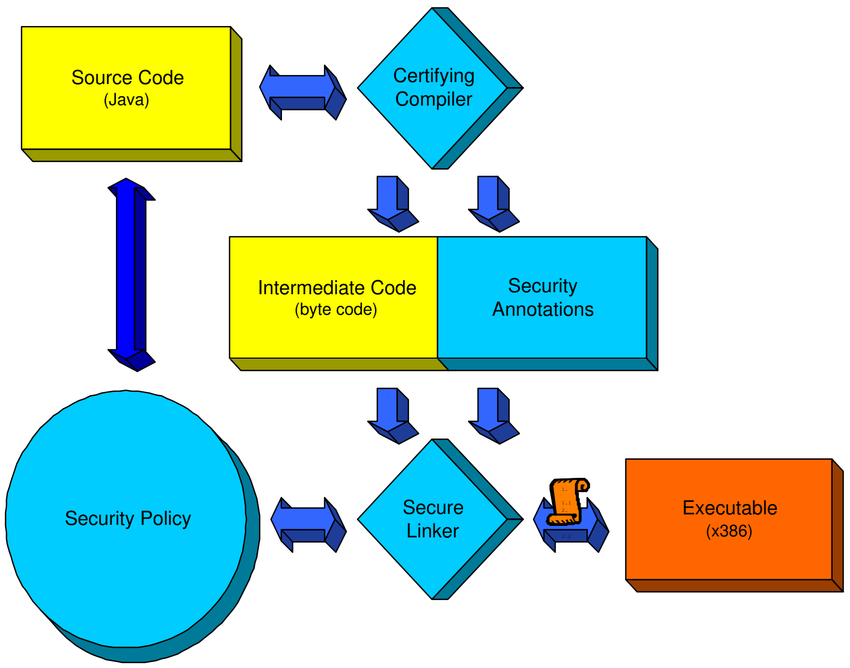

Figure 1: A certifying compiler

The main difficulties encountered with certifying compilers are associated with mathematical complexity which grows rapidly with the size of the software component, sometimes to an unmanageable level. A survey of research efforts around the world [10] indicates that a pragmatic approach to deal with the certification complexity should be based on the modularization of software. When such modules are assembled to form a large system, this should be associated with a formal composition process that includes rigorous security management.

3 Key Lessons Learned from the MaliCOTS Project

As mentioned above, each technique has its own strengths and weaknesses and they are summarized in Table 1. The MaliCOTS project provided a better understanding of where and when specific tools should be used for a particular purpose. Table 2 gives a brief overview of the prototype tools that were developed to detect malicious code in COTS software.

Additionally, further general conclusions about software certification have been derived from the MaliCOTS project. Key lessons learned were:

i Security management should begin as early as possible in the design phase of critical systems

  (a) To modularize the C2IS with respect to security enforcement

  (b) To avoid retro-fitting security in "canned" systems

ii Security constraints should be expressed explicitly in the C2IS model

iii Preliminary certification for coherence and completeness before engaging in source-code generation is highly desirable.

Table 1: Basic Comparison of Certification Techniques

Technique:	**Good for:**	**Limited by:**
Execution Monitoring	• Live access control • Concurrent processes • User surveillance • Viruses and worms interceptor • Covert channel detection • Monitoring of temporary files	• Reactive in nature; may be overridden by execution • May slow down system response • It is difficult to keep track of complex behaviours • Must be focussed to perform efficiently
Static Analysis	• Detection of dead code • Control flow logic and slicing • Data flow analysis • Detection of hidden functionality • Program visualization and understanding	• Complexity grows rapidly with software size • Very difficult on binary executable without source code • Highly-skilled expertise mandatory • Copyright may preclude analysis • Obfuscators • Some properties are unverifiable
Certifying Compiler	• Memory safety • Control flow safety • Data flow analysis • Type safety • Mobile code risk management	• Novel and under-estimated technique • Certification complexity grows rapidly with module size • Security policy must be rigorously expressed • Some properties are unverifiable

Since most certification techniques that are applicable to source code lead to a rapid increase in the certification complexity, it appears mandatory to modularize large systems into small components that are independently certifiable. The composability of these modules must be examined in order to make sure that a large system built from trusted components is still secure.

Also, an often forgotten requirement is that a security policy must be rigorously expressed to be enforceable. It must be possible to express this policy at many levels of abstraction so as to cope with the wide spectrum of security requirements. The granularity of security constraints ranges from high-level rules (e.g. no information leakage), to low level ones (e.g. all temporary files must be deleted on exiting a program). As stated above, security policies must be applicable to modular software. Despite many research activities identified in a state-of-the-art paper [13], it appears that software security modelling is lagging behind the development of certification tools.

In summary, security management must begin at the design stage so as to save time and money

Table 2: MaliCOTS Prototypes

Technique:	**Prototype Tool:**	**Key Concepts:**
Execution Monitoring	DaMon	• Wrapping of the OS • Applications require services through an instrumented OS – port surveillance (e.g. communication control) – file monitoring (e.g. secrecy enforcement) – process monitoring (e.g. prevent denial of service) – OS configuration database (e.g. Registry in Windows)
Static Analysis	SamCOTS	• Program translation into an abstract notation • OS API checks (e.g. basic OS functions verification) • Pattern recognition of complex malicious behaviour via their generic description in Schneider-inspired automata
Certifying Compiler	TALCC	• ANSI C certifying compiler • Based on typed assembly language • Safety annotations are compared with a security policy • Model-checking hidden in an easy-to-use tool
	JACC	• Java certifying compiler • Based on an extension to the Java-type system • Requires an augmented Java verifier to ensure memory, control flow, and type safety

and to modularize large C2IS in a structured manner. This will allow for the assembly of a large system out of independently-certified modules. This is the focus of the current research, which is described in the next section.

4 Current R&D Challenges: Modelling Secure Systems

UML is currently the prime notation for C2IS-modelling in Canada. It is also a de facto standard notation in the software industry. UML has already been proven useful for documentation and for communication between developers, etc.. In order to build on existing expertise and common practice, it was decided to extend UML to include security constraints.

4.1 Why Object Constraint Language (OCL)

In 1998, OCL already appeared to be a good approach [7] for increasing the preciseness and quality of a C2IS model.

Object Constraint Language (OCL) is a notational language proposed by the Object Management Group (OMG) to specify constraints (invariants, pre/post-conditions) over object models. Three studies were carried out to confirm the potential of OCL. First, Laurendeau et al. [3] investigated the technical foundations and confirmed the usefulness of OCL in reducing ambiguities in UML diagrams by specifying constraints such as pre-/post- conditions and invariants.

As a result of this, it was decided to evaluate further the cost and benefits of using OCL in the modelling of C2IS. In 2000, a C2IS that had been modelled previously in UML in a DRDC Valcartier project, was ‘enhanced’ with OCL constraints. Dessureault et al. [7] showed that OCL is not only useful in improving reliability, but it is also relatively inexpensive to include in UML diagrams. It was estimated that the additional investment needed to include OCL expressions would be only about 10% to 15% if it were done while defining the UML diagrams of C2IS.

The expressiveness of OCL was then carefully examined in order to make sure that this notation could express not only reliability constraints but also security constraints. In [14], the key issue was analyzed and it confirmed that OCL is a sustainable and efficient technology to improve reliability and security in C2IS modelling.

4.2 How UML/OCL Diagrams can be Certified

These positive evaluations of OCL led to the creation of the SOCLe project in 2002. The objective is to demonstrate how C2IS certification can start at the design phase, using UML diagrams to model the system and OCL as a property specification language.

The technical approach involves two main steps [15]:

1. Translate security rules into UML design models (via OCL Constraints), and

2. Check for compliance (via model-checking of OCL security constraints.)

In practice, the architect designs his C2IS using UML in the usual way. He makes his description as precisely as possible using state-charts, collaboration diagrams, class diagrams and OCL constraints. While he is adding syntactic constructs to his description, the design-time verification tool constructs the underlying model that is formally checked. The process is transparent to the designer who requires only a basic awareness of it to do his work and the complexities of formal methods are completely hidden from him.

Since restrictions inherent to graphical diagrams cannot cover the entire behaviour of the system and since some properties cannot be verified until source code is written, it is not expected that the whole risk can be manageable at the design phase. As previously mentioned, some types of risk can be managed only at compile time or even during execution. However, useful results can be obtained by certifying the model. For example:

- Notes of conformity for the managed risk at the model level;

- Source code and/or associated annotations for the risk that needs to be managed by the certifying-compiler;

- Indications of what should be monitored during execution; and
- Test sets for the residual risk that is unmanageable by other means.

Thus, as the information-processing system takes shape, the compliance certificate is enriched by notes, which ultimately ensures the enforcement of a large part of the security policy in the development process (from design to source code.) Also, this process provides a precise identification of the residual risk that has not been managed. This residual risk can then be managed by an execution monitor or guaranteed against by the use of exhaustive but targeted tests.

A rudimentary prototype was developed during the summer of 2002 at l'École Polytechnique de Montréal. It confirmed the feasibility of model-checking abstracted UML/OCL diagrams [2]. A thorough literature survey [16] also enumerates research activities from around the world on related techniques. It provides a comprehensive overview of this field. For readers interested in seeing some of this UML/OCL syntax, Annex A presents the UML diagrams of a C2IS component that was enriched with OCL constraints as part of an early evaluation of OCL in the C2IS context [7].

In parallel with the SOCLe project, the feasibility of developing a complete set of secure design patterns is currently being studied. This could greatly ease secure system development and could be used as a case study for the certification prototype.

5 Discussion and Conclusions

Over the past decade, a great deal of progress has been made in the certification of critical systems. Many large software editors are now making serious commitments toward the production of more reliable and secure software. The impressive success of Open-Source software is largely a result of the excellent reputation that these products have garnered in terms of quality and reliability. It is expected that the need for secure and reliable software will continue to increase with the proliferation of e-commerce and other applications requiring high-confidence software.

After five years of research in software certification, it appears mandatory to associate the system design closely with the security specification. Security enforcement must start at the design phase not only to save time and money but also to organize its implementation among the various engines involved in the process, each with its own strengths and weaknesses.

It is quite likely that the way in which C2IS are modularized will have to evolve with the demands for managing the intrinsic and extrinsic security of each component. This is the key feature to control the state-space growth that has been the traditional pitfall of formal methods for the past twenty years.

References

[1] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M. Erhioui, Yvan Lavoie, and Nadia Tawbi. Static Detection of Malicious Code in Executable Programs. In *Symposium on Requirements Engineering for Information Security (SREIS)*, Purdue University, Indianapolis, ID, USA, March 2001. Center for Education and Research in Information Assurance and Security (CERIAS). `http://www.cerias.purdue.edu/SREIS.html`. 4

[2] Mathieu Bergeron, Gaétan Hains, John Mullins, and Martin Salois. Building an OCL Constraint Checker. In *Foundations of Computer Security*, Ottawa, Ontario, Canada, June 2003. IEEE. LICS'03 Satellite Workshop, Submitted. 9

[3] Louis Borgeat, Denis Dion, Antoine Escobar, Denis Laurendeau, and Marielle Mokhtari. Systèmes d'information répartis. Contract report, Defence Research Establishment Valcartier (DREV), Val-Bélair, Québec, Canada, December 1999. 8

[4] Robert Charpentier and Martin Salois. Secure Software from Design to Binary. In *High Confidence Software and Systems Conference*, Baltimore, Maryland, United States, April 2003. NSA. 4

[5] Robert Charpentier, Martin Salois, Jean Bergeron, Mourad Debbabi, Jules Desharnais, Emmanuel Giasson, Béchir Ktari, Frédéric Michaud, and Nadia Tawbi. Secure Integration of Critical Software via Certifying Compilers. In *Information Technology Security Symposium (CSE-CITSS)*, Ottawa, Canada, June 2001. Canadian Security Establishment (CSE), (CSE). 4

[6] Mourad Debbabi, Marc Girard, Luc Poulin, and Martin Salois. Monitoring of Malicious Activity in Software Systems. In *Symposium on Requirements Engineering for Information Security (SREIS)*, Purdue University, Indianapolis, ID, USA, March 2001. Center for Education and Research in Information Assurance and Security (CERIAS). `http://www.cerias.purdue.edu/SREIS.html`. 3

[7] Dany Dessureault and Richard Caron. Software Reuse in Command and Control: Initial UML/OCL Framework. Contract report, Defence Research Establishment Valcartier, Québec, Canada, March 2000. 8, 9, 12

[8] Richard Hunter, Arabella Hallawell, and Neil MacDonald. Into the Era of Mass Victimization. Gartner, Inc., `http://www.gartner.com`, September 2001. Research Note, Strategic Planning, SPA-12-9385. 2

[9] Joost-Pieter Katoen. *Concepts, Algorithms, and Tools for Model Checking*. Lehrstuhl für Informatik VII Friedrich-Alexander Universität Erlangen-Nürnberg, 1998. Lecture Notes of the Course "Mechanised Validation of Parallel Systems". 2

[10] François Kirouac (CGI) and Martin Salois. The State of the Art in Certifying Compilers. Contract report, Defence Research & Development Canada — Valcartier, Val-Bélair, Québec, Canada, July 2002. 5

[11] François Kirouac (CGI) and Martin Salois. The State of the Art in Dynamic Analysis. Contract report, Defence Research & Development Canada — Valcartier, Val-Bélair, Québec, Canada, July 2002. 3

[12] François Kirouac (CGI) and Martin Salois. The State of the Art in Static Analysis. Contract report, Defence Research & Development Canada — Valcartier, Val-Bélair, Québec, Canada, July 2002. 4

[13] Béchir Ktari (Université Laval). The State of the Art in the Specification of Security Policies. Contract report, Defence Research & Development Canada — Valcartier, Val-Bélair, Québec, Canada, March 2003. 6

[14] Vincent Labbé and Marc Girard. Mitigating the Insider Threat to Information Systems: Tools, Policies and UML/OCL. Contract report, Defence Research Establishment Valcartier, Québec, Canada, June 2001. 8

[15] John Mullins and Mathieu Bergeron. SOCLe project: Model-checking OCL Constraints to secure UML Models (Theoretical Study). Contract report, Defence Research & Development Canada Valcartier, Québec, Canada, September 2002. 8

[16] John Mullins, Mathieu Bergeron, and Geneviève Bastien. SOCLe project: Adapting Model-Checking Techniques to Secure UML/OCL Models (Litterature Review). Contract report, Defence Research & Development Canada Valcartier, Québec, Canada, September 2002. 9

Acknowledgements

The authors wish to thank the twenty-or-so graduate and undergraduate students who contributed to these research projects over the past five years. Special thanks are due to Prof. John Mullins (Team leader at l'École Polytechnique de Montréal) and Prof. Mourad Debbabi (Team leader at l'Université Laval) for their exceptional commitment to our research program. Thanks are also due to Brian Cheers for his patient review of our articles.

Annex A

C2IS Design With OCL Constraints[1]

In order to evaluate the feasibility of integrating OCL into each phase of the software lifecycle, a small-scale study was performed in 1999-2000. The Case_Atti (Concept Analysis and Simulation Environment for Automated Target Tracking and Identification) test-bed was selected as a representative example of C2IS and it is presented in the first part of this Annex. The results of applying OCL to the Case_Atti test-bed are given below in tabular form.

A.1 Case_Atti Test-bed

The software system Case_Atti was largely designed and modelled using UML notation. Also there is a one-to-one relationship with the code for many classes of the model. A large part of Case_Atti was developed using the CASE-tool Rational Rose. Finally, Case_Atti is a system which is still in use at DRDC Valcartier. This justifies the choice of a sample of Case_Atti as the subject for the OCL test-bed.

A.1.1 Scope of the Demonstration

The objective of the test-bed was to validate the applicability of a Design-by-contract (DBC) approach based on OCL to a typical C2IS. This meant that it was necessary to specify the subject model, to implement DBC as it should have been initially implemented, to integrate OCL constraints derived from contracts into the model and finally to propagate the constraints into the actual code.

The model of the Case_Atti data filters was identified as a good candidate for the test-bed. More precisely, the class IMMFilterClass and the classes it interacts with were analyzed to apply DBC and to integrate OCL constraints. Figure 2 shows the model of the class IMMFilterClass and its main relationships with its surrounding classes. The class IMMFilterClass and its surrounding classes are used to model an approach to handle target manoeuvres within single- and multiple-target-tracking systems. More precisely, the IMMFilterClass models the IMM (Interacting Multiple Model) filter. In the multiple-model approach, it is assumed that the system state will adhere to one of a finite number of models or modes. Each model is characterized by its own particular parameters.

The class IMMFilterClass is a good candidate for applying DBC and OCL since it has to interact with many other classes in order to fulfil the expected services for which it was designed. There is multiple inheritance between the class IMMFilterClass and the classes IMMFilterInitClass and the FilterClass. The class IMMFilterClass has to interact with the aggregates FilterListClass and LikelihoodFunctionEvaluatorClass to produce the majority of the algorithms within the IMM filter model. The class IMMFilterClass also has many other interactions with the classes it depends upon, namely the MatrixArrayByValClass, the SystemTrackIntIndexListClass, the ResidualListClass and the ResidualClass.

[1]Excerpt adapted from a contract report by Dessureault and Caron [7]

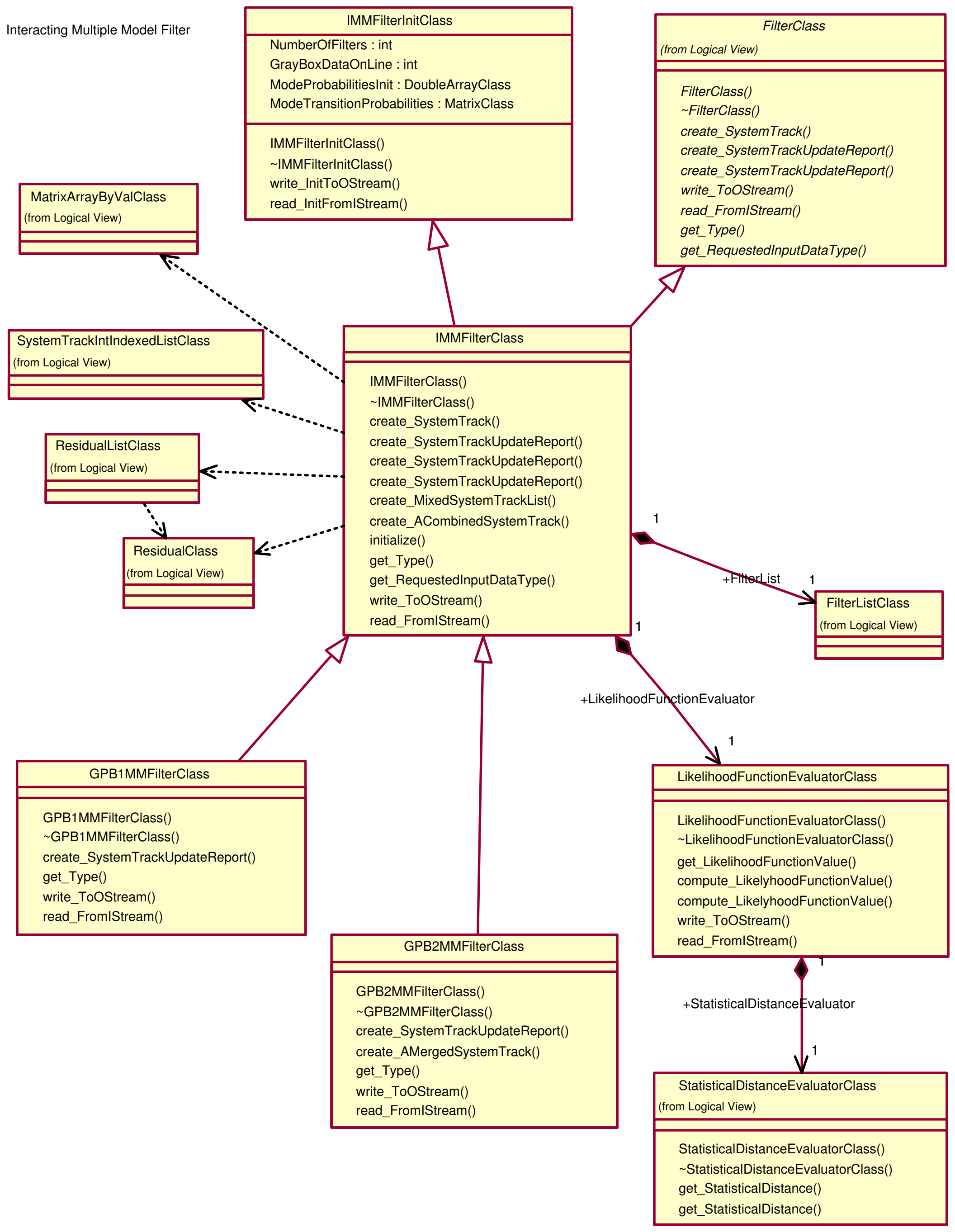
Interacting Multiple Model Filter
IMMFilterInitClass
NumberOfFilters : int
GrayBoxDataOnLine : int
ModeProbabilitiesInit : DoubleArrayClass
ModeTransitionProbabilities : MatrixClass
IMMFilterInitClass()
~IMMFilterInitClass()
write_InitToOStream()
read_InitFromIStream()
FilterClass
(from Logical View)
FilterClass()
~FilterClass()
create_SystemTrack()
create_SystemTrackUpdateReport()
create_SystemTrackUpdateReport()
write_ToOStream()
read_FromIStream()
get_Type()
get_RequestedInputDataType()
MatrixArrayByValClass
(from Logical View)
SystemTrackIntIndexedListClass
(from Logical View)
ResidualListClass
(from Logical View)
ResidualClass
(from Logical View)
IMMFilterClass
IMMFilterClass()
~IMMFilterClass()
create_SystemTrack()
create_SystemTrackUpdateReport()
create_SystemTrackUpdateReport()
create_SystemTrackUpdateReport()
create_MixedSystemTrackList()
create_ACombinedSystemTrack()
initialize()
get_Type()
get_RequestedInputDataType()
write_ToOStream()
read_FromIStream()
1
+FilterList
1
FilterListClass
(from Logical View)
1
+LikelihoodFunctionEvaluator
1
GPB1MMFilterClass
GPB1MMFilterClass()
~GPB1MMFilterClass()
create_SystemTrackUpdateReport()
get_Type()
write_ToOStream()
read_FromIStream()
GPB2MMFilterClass
GPB2MMFilterClass()
~GPB2MMFilterClass()
create_SystemTrackUpdateReport()
create_AMergedSystemTrack()
get_Type()
write_ToOStream()
read_FromIStream()
LikelihoodFunctionEvaluatorClass
LikelihoodFunctionEvaluatorClass()
~LikelihoodFunctionEvaluatorClass()
get_LikelihoodFunctionValue()
compute_LikelyhoodFunctionValue()
compute_LikelyhoodFunctionValue()
write_ToOStream()
read_FromIStream()
1
+StatisticalDistanceEvaluator
1
StatisticalDistanceEvaluatorClass
(from Logical View)
StatisticalDistanceEvaluatorClass()
~StatisticalDistanceEvaluatorClass()
get_StatisticalDistance()
get_StatisticalDistance()

Figure 2: Model of the Class IMMFilterClass

To give a good idea of the program size required to effect the modelling of the IMM filter, Table 3 shows the number of lines of C++ code (including the comments) for the class IMMFilter-Class and its surrounding classes. It is to be noted that the “list” classes like the ResidualListClass

Table 3: Number of Lines of Code for the classes of the IMM Filter Model

Class of the IMM Filter Model	Number of Lines of Code
IMMFilterClass	1114
FilterClass	223
IMMFilterInitClass	213
FilterListClass	119
LikelihoodFunctionEvaluatorClass	290
StatisticalDistanceEvaluatorClass	201
MatrixArrayByValClass	83
SystemTrackIntIndexedListClass	118
ResidualListClass	118
ResidualClass	229
ListClass	1458
Total for all classes of the IMM Filter Model	4166

are “parameterized” classes that perform the instantiation of the template class ListClass. It should also be noted that the derived classes GPB1MMFilterClass and GPB2MMFilterClass were not considered as part of the test-bed.

A.1.2 Derived Contracts & Constraints

As mentioned above, one of the first major tasks of the test-bed was to identify OCL Constraints related to the class IMMFilterClass which was chosen as the scope for the test-bed.

These OCL constraints are referred to as "derived" constraints because they were obtained a posteriori by analyzing the existing code (reverse-engineering) and also from discussions with the individuals involved in the initial creation.

Above all, constraints are obtained by answering two questions:

1. what are the rules that ensure the integrity of an object at all times? and
2. what are the conditions which control the execution of an object’s operation?

Apart from stating them, no formal approach is required to answer the first question (object’s integrity rules). The second question, though, must be addressed using a formal approach such as DBC. By describing an object’s collaboration with other objects within contracts, one can identify the responsibilities of each party and therefore derive the corresponding constraints.

Table 4 presents a representative set of contracts where the class IMMFilterClass is involved. Since the process was partly performed by reverse-engineering, each public method was considered as a potential contract with the external world and the name of the operation is shown in parentheses.

Table 5 presents the list of the constraints applied to the class (or parts of) IMMFilterClass.

Table 4: Representative Set of Contracts Where IMMFilterClass is Involved

Client Object	**Supplier Object**	**Contract (Operation)**	**Client**	**Supplier**
FilterManager	IMMFilterClass	Create a new SystemTrack for a contact (create_SystemTrack())	Obligations ✓ Provide a contact which cannot be assigned to any other SystemTrack. Benefits ➢ Obtain a new SystemTrack.	Obligations ✓ Create and return a new SystemTrack that has no speed value and contains a single contact, the one provided by the client. This SystemTrack is called a “no-speed” SystemTrack. Benefits ➢ No need to create a SystemTrack if no contact provided.
		Update a SystemTrack with a new contact (create_SystemTrackUpdateReport() #1)	Obligations ✓ Provide a contact ✓ Provide a VALID and COMPATIBLE SystemTrack ✓ VALID systemTrack means that the type of the SystemTrack provided must be one of the following: MultipleXYVxVySystemTrackClass OR XYSystemTrackClass ✓ COMPATIBLE SystemTrack means that the number of unitary tracks contained in the SystemTrack is equal to the number of simple filters contained in this instance of the IMMFilterClass. Benefits ➢ Obtain a regular SystemTrack which has been updated and has a speed calculated for the assigned contact.	Obligations ✓ Update the SystemTrack with the contact provided by the client. ✓ Calculate the speed of the SystemTrack. ✓ Return a regular SystemTrack Benefits ➢ No need to update SystemTrack if no contact provided. ➢ No need to update SystemTrack if no SystemTrack provided or if the SystemTrack is not valid or compatible.
IMMFilterClass (*)	IMMFilterClass (*)	Update a “no speed” SystemTrack with a new contact (initialize())	Obligations ✓ Provide a contact ✓ Provide a “no speed” Valid and Compatible SystemTrack Benefits ➢ Obtain a regular SystemTrack which has been updated and has a speed calculated for the assigned contact.	Obligations ✓ Update the “no speed” SystemTrack with the contact provided by the client ✓ Calculate the speed of the SystemTrack. ✓ Return a SystemTrack. Benefits ➢ No need to update the SystemTrack if no contact provided. ➢ No need to update SystemTrack if no SystemTrack provided , or if the SystemTrack is not valid or compatible.

Table 4: Representative Set of Contracts Where IMMFilterClass is Involved (continued)

Client Object	**Supplier Object**	**Contract (Operation)**	**Client**	**Supplier**
		Update a regular SystemTrack with a new contact. (create_SystemTrackUpdateReport() #2)	Obligations ✓ Provide a contact ✓ Provide a Valid and Compatible regular SystemTrack. ✓ Sum of mode probabilities (MU) of the SystemTrack must be equal to 1 Benefits ➢ Obtain a regular SystemTrack which has been updated and has a speed re-calculated for the assigned contact.	Obligations ✓ Update the regular SystemTrack with the contact provided by the client ✓ Calculate the speed of the SystemTrack. ✓ Return a regular SystemTrack Benefits ➢ No need to update SystemTrack if no contact provided. ➢ No need to update SystemTrack if no SystemTrack provided or if the SystemTrack is not valid or compatible. ➢ No need to update the SystemTrack if the sum of mode probabilities not equal to 1.

(*). Usually, a contract involves different parties since its purpose is to describe interactions between distinct objects. In this special case, Client and Supplier are the same because it was decided to take advantage of the process to further document the IMMFilterClass operations.

Table 5: List of Constraints Applied to the Class IMMFilterClass

Operation	**Constraints**			
	Type[1]	**Description**	**OCL Expression**[2]	**C++ Assert Expression**
(not applicable for Invariant)	Inv	Filter list contains at least 1 filter.	self.FilterList.get_Collection( )->size >= 1	assert(FilterList.get_Cardinality() >= 1);
	Inv	All filters contained in the filter list are of the same type. To simplify the test, all filters should have the same type as the first in the collection.	self.FilterList.get_Collection( )->forAll (oclType = self.FilterList.get_Collection( )->first)	FilterIter.first(); First_Type = FilterIter.get_Current()->get_Type(); for(FilterIter.next(); FilterIter.is_NotAtEnd(); FilterIter.next()) {assert(FilterIter.get_Current()->get_Type() = = First_Type);}
	Inv	Type of all filters contained in the filter list are of simple type (StandardKalmanFilterClass or ConvertedMeasurementKalmanFilterClass)	self.FilterList.get_Collection( )->forAll ((oclType.name = 'StandardKalmanFilterClass') or (oclType.name = 'ConvertedMeasurementKalmanFilterClass'))	for(FilterIter.first(); FilterIter.is_NotAtEnd(); FilterIter.next()) {assert(FilterIter.get_Current->get_Type() = = STANDARD_KALMAN_FILTER_TYPE || FilterIter.get_Current->get_Type() = = CONVERTED_KALMAN_FILTER_TYPE); }
	Inv[3]	For all lines of Pij Matrix, the sum of the column values is greater than 0	self.ModeTransitionProbabilities.get_Lines()->forAll(line: Collection | line->sum >0)	for(i=1; i<=Pij.get_LineDim(); i++) {sum = 0.; for(j=1; j<=Pij.get_ColumnDim(); j++) {sum = sum + Pij(i,j);}assert(sum>0.);}
	Inv[3]	For all columns of Pij Matrix, the sum of the line values is greater than 0	self.ModeTransitionProbabilities.get_Columns()->forAll(column: Collection | column->sum >0)	for(j=1; j<=Pij.get_ColumnDim(); j++) {sum = 0.; for(i=1; i<=Pij.get_LineDim(); i++) {sum = sum + Pij(i,j);}assert(sum>0.);}
create_SystemTrack(InpData:InputDataElementClass&, SystemTrackID:SystemTrackIDClass,...) :SystemTrackClass *	Pre	Contact provided is not Null	InpData.notNil	assert(&InpData != NULL);
	Post	SystemTrack created and returned has no speed.	SystemTrack.oclType.name = 'XYSystemTrackClass'	assert(SystemTrack->get_Type() = = XY_TRACK_TYPE);
create_SystemTrackUpdateReport(InpData:InputDataElementClass&, SysTrack:SystemTrackClass&, TimeAlig:TimeAlignerParameterClass&, SystemTrackUpdateReportMask:int, SystemTrackID:SystemTrackIDClass, ToInformationBag:NonComputationPurposeInformationBagClass *) :SystemTrackUpdateReportClass *	Pre	Contact provided is not Null	InpData.notNil	assert(&InpData != NULL);
	Pre	The type (class) of the SystemTrack provided can only be one of these 2 types: MultipleXYVxVySystemTrackClass OR XYSystemTrackClass	(SystemTrack.oclType.name = 'MultipleXYVxVySystemTrackClass') or (SystemTrack.oclType.name = 'XYSystemTrack')	assert(SysTrack->get_Type() = = MULTIPLE_XY_VX_VY_TRACK_TYPE || SysTrack->get_Type() = = XY_TRACK_TYPE);
	Pre	Number of unitary tracks in the SystemTrack is equal to the number of simple filters contained in this IMMFilter	self.FilterList.get_Collection( )->size = SysTrack.SYVxVySystemTrackList->get_Collection()->size	assert((MultipleXYVxVySystemTrackClass &)SysTrack->get_NumberOfTracks() = = FilterList.get_Cardinality());
	Post	SystemTrack updated and returned must have a speed.	SystemTrack.oclType.name <> 'XYSystemTrackClass'	assert(SystemTrack->get_Type() != XY_TRACK_TYPE);
initialize(InpData:InputDataElementClass&, XYSysTrack:XYSystemTrackClass&, TimeAlig:TimeAlignerParameterClass&, SystemTrackUpdateReportMask:int, SystemTrackID:SystemTrackIDClass, ToInformationBag:NonComputationPurposeInformationBagClass*) :SystemTrackUpdateReportClass *	Pre	Contact provided is not Null	InpData.notNil	assert(&InpData != NULL);
	Pre	The type (class) of the SystemTrack provided can only be one of these 2 types: MultipleXYVxVySystemTrackClass OR XYSystemTrackClass	(SystemTrack.oclType.name = 'MultipleXYVxVySystemTrackClass') or (SystemTrack.oclType.name = 'XYSystemTrack')	assert(SysTrack->get_Type() = = MULTIPLE_XY_VX_VY_TRACK_TYPE || SysTrack->get_Type() = = XY_TRACK_TYPE);
	Pre	Number of unitary tracks in the SystemTrack is equal to the number of simple filters contained in this IMMFilter	self.FilterList.get_Collection( )->size = SysTrack.SYVxVySystemTrackList->get_Collection()->size	assert((MultipleXYVxVySystemTrackClass &)SysTrack->get_NumberOfTracks() = = FilterList.get_Cardinality());
	Pre	SystemTrack provided has no speed.	SystemTrack.oclType.name = 'XYSystemTrackClass'	assert(SystemTrack->get_Type() = = XY_TRACK_TYPE);
	Post	SystemTrack updated and returned must have a speed.	SystemTrack.oclType.name <> 'XYSystemTrackClass'	assert(SystemTrack->get_Type() != XY_TRACK_TYPE);

Table 5: List of Constraints Applied to the Class IMMFilterClass (continued)

Operation	Constraints			
	Type[1]	Description	OCL Expression[2]	C++ Assert Expression
create_SystemTrackUpdateReport(InpData:InputDataElementClass&, MultipleXYVxVySystemTrack:MultipleXYVxVySystemTrackClass&, TimeAlig:TimeAlignerParameterClass&, SystemTrackUpdateReportMask:int, SystemTrackID:SystemTrackIDClass, ToInformationBag:NonComputationPurposeInformationBagClass*) :SystemTrackUpdateReportClass *	Pre	Contact provided is not Null	InpData.notNil	assert(&InpData != NULL);
	Pre	The type (class) of the SystemTrack provided can only be one of these 2 types: MultipleXYVxVySystemTrackClass OR XYSystemTrackClass	(SystemTrack.oclType.name = 'MultipleXYVxVySystemTrackClass') or (SystemTrack.oclType.name = 'XYSystemTrack')	assert(SysTrack->get_Type() = = MULTIPLE_XY_VX_VY_TRACK_TYPE || SysTrack->get_Type() = = XY_TRACK_TYPE);
	Pre	Number of unitary tracks in the SystemTrack is equalto the number of simple filters contained in this IMMFilter	self.FilterList.get_Collection()->size = SysTrack.SYVxVySystemTrackList->get_Collection()->size	assert((MultipleXYVxVySystemTrackClass &)SysTrack->get_NumberOfTracks() = = FilterList.get_Cardinality());
	Pre	Sum of all MUi (Mode probabilities) of the SystemTrack must be equal to 1	(SystemTrack.get_MMFilterAllocation().get_ModeProbabilities()->sum) = 1	assert(sum = 0.; for(i=1; i<=MUi.get_Cardinality(); i++) {sum = sum + MUi(i);} assert(i = = 1.);
	Post	SystemTrack updated and returned must have a speed.	SystemTrack.oclType.name <> 'XYSystemTrackClass'	assert(SystemTrack->get_Type() != XY_TRACK_TYPE);

(1) Types are: invariant (Inv), pre -conditions (Pre) and post-conditions (Post).
(2) They are also presented in OCL Constraints file.
(3) These invariants are inherited from IMMFilterInitClass.