SOCLe: Integrated Design of Software Applications and Security

Frédéric Painchaud
Defence Research and Development Canada – Valcartier
2459 Pie-XI Blvd North
Val-Bélair (Québec)
Canada, G3J 1X5
Frederic.Painchaud@drdc-rddc.gc.ca

Damien Azambre
Mathieu Bergeron
John Mullins
Raveca Maria Oarga
Department of Computer Engineering
École Polytechnique de Montréal
P.O. Box 6079, Centre-ville Station
Montréal (Québec)
Canada, H3C 3A7
{damien.azambre, mathieu-2.bergeron,
john.mullins, raveca-maria.oarga}@polymtl.ca

Abstract

Defence Research and Development Canada (DRDC) – Valcartier, with the support of the CRAC Laboratory at École Polytechnique de Montréal, carried out an ambitious R&D project aiming at developing a tool called SOCLe which integrates the design of software applications with their security. Integrating the design of software applications and security into a unique tool is not only particularly innovative but also crucial to ensure the quality and security of critical command and control information systems. Moreover, as it is 50 to 100 times more expensive to correct a software error once the design phase is finished, detecting and correcting errors at design-time saves time and money. This paper presents the SOCLe project’s major achievements along with a simplified case study on Caveats, as an illustrative example of a possible military application for this new technology.

1. Introduction

It is now generally recognized that the design phase is crucial to the development of high-quality software. Nowadays, software is omnipresent and its faults can involve enormous social and economic repercussions. Sad examples abound: the Therac-25 radiotherapy machine (3 dead and 3 severely wounded persons), the ARIANE 5 rocket (approximately 500 million US dollars loss), the breakdown of AT&T network (approximately 60 million US dollars loss), etc.\footnote{According to wikipedia.org and cnn.com.} However, programs’ behavior is generally only validated after their partial or complete implementation, using various testing techniques. Moreover, it is estimated that correcting a software error after its implementation is 50 to 100 times more expensive than at design-time [2]. Thus, there is a real need for design-time validation techniques.

With this objective in mind, Defence Research and Development Canada (DRDC) – Valcartier initiated a project to develop a tool that allows the integration of security within the design of software applications. The idea is an innovative exploitation of constraints in Unified Modeling Language (UML) diagrams. These security constraints are expressed in the Object Constraint Language (OCL), hence the name Secure OCL expressions (SOCLe). The UML notation is currently the de facto industrial standard for the design of object-oriented, modern software. Indeed, it is used to support software engineering best practices. With UML, it is possible to specify the structural and behavioral aspects of software using standardized diagrams. OCL is integrated into the UML standard and expresses class invariants and pre/post-conditions on operations. These constraints specify and clarify the meaning of UML diagrams by precisely describing the correct model behavior.

The SOCLe project is carried out by a team of student members of the Conception et Réalisation des Applications Complexes (CRAC) Laboratory at École Polytechnique de Montréal, and is directed by professor John Mullins. It tackles the important problem of design-time validation. In
the first research phase, the SOcLe team compiled an
exhaustive state-of-the-art study in order to judiciously choose
technologies to be used throughout the project. UML was
essentially selected because of its popularity in the industry.
OCL is interesting for several reasons: it is integrated
into the UML standard, it is easy to use, and, especially, it
is possible to give it a precise and unambiguous meaning
and to integrate it into rigorous verification techniques. Re-
search efforts continued thereafter with the formal design
necessary for the implementation of the SOcLe tool. This
implementation phase initially targeted the development of
an automatic verification engine, and then the design of a
graphical user interface for this engine. Considerable ef-
forts were also invested in order to improve the engine’s
performance, which is now adequate.

The rest of this paper is structured as follows. Section
2 discusses similar tools found in the literature. Section 3
presents the general architecture of the SOcLe tool. Section 4
details the simplified case study used throughout this pa-
per: a caveat-separation system. Sections 5, 6, and 7 sketch
the utilization of the tool using this case study. Finally, Sec-
tion 8 presents ongoing improvements to the tool and con-
cclusions. Note that some familiarity with UML and OCL is
assumed. The reader is referred to [10] for an introduction
to UML. Details about OCL can be found in [9].

2. Related Work

Few research tools are currently available for static
and/or dynamic validation of UML diagrams. Lilius and
Paltor [8] propose vUML, a tool that translates UML state-
chart diagrams into Promela, the modeling language of the
SPIN model-checker. Constraints are expressed through
Linear Temporal Logic (LTL) and are transmitted to SPIN.
Similarly, Latella et al. [7] give a provably-correct transla-
tion from a subset of UML statechart diagrams into Promela.
None of these approaches, however, have taken OCL into
account.

The toolset proposed by Shen et al. [11] offers static
dynamic validation. Static checks include: syntactic
correctness according to well-formedness constraints of the
UML meta-model, coherence of an object diagram with re-
spect to a class diagram, and validation of OCL constraints
on an object diagram. The dynamic checks consist of the
verification of statechart diagrams using the SMV model-
checker. However, OCL constraints are not considered by
this model-checker either. Therefore, they are not supported
as a dynamic constraint specification language.

Extensions for OCL constraints are suggested by Diste-
fano et al. [5], using Computational Tree Logic (CTL), and
Bradfield et al. [3], using propositional \( \mu \)-calculus. The
general idea is to replace the atomic properties of these tem-
poral logics by standard OCL constraints. Note, however,
that neither of these frameworks support UML as a design
notation.

3. SOcLe Architecture

SOcLe is divided into three main modules:

1. a UML compiler,
2. a specialized Abstract State Machine (ASM) interpreter
   [1], and
3. a model-checker (Figure 1).\footnote{For the sake of brevity and to respect the scope of this paper, no tech-
nical details are given about the ASM formalism and model-checking tech-
niques.}

The tool works with UML diagrams expressed in the
XML Metadata Interchange (XML) format, which is sup-
ported by most UML CASE tools.

The verification process has three phases:

1. translate the UML diagrams into an ASM executable
   specification,
2. simulate the execution of the generated specification, and
3. verify OCL constraints against the resulting execution
   graph.

The UML compiler translates UML diagrams to an ASM
giving a formally-defined semantics. Basic diagram
elements, such as classes and method names, are mapped
to sorts (data domains). More complex elements, such as
method declarations and statechart transitions, are trans-
lated into enumerated functions. The object diagram is
mapped to a specific subset of these functions and repres-
ents the initial configuration of the UML diagrams.

The ASM interpreter simulates the execution of the ASM
specification. From an initial configuration, successor con-
figurations are computed by evaluating an ASM program
capturing the dynamic behavior of the UML diagrams.

The model-checker verifies the OCL constraints by tran-
slating them into a set of properties and applying a verifi-
cation algorithm [4]. The result is transmitted back to the
graphical user interface through a set of diagnostic files.

SOcLe includes a graphical user interface embedded into
argUML, a customizable open-source UML CASE tool\footnote{http://argouml.tigris.org.}. It
allows the designer to visualize verification results and to
inspect the diagrams’ execution graph.
4. Caveat-Separation System: Simplified Case Study Specifications

The main objective of the case study is relatively simple: design and verify a simplified caveat-separation system based on the following specifications.

4.1. Users

Of course, a caveat-separation system implies the management of users. In this system, users have the following attributes:

- A country of origin. For example: “Canada”, “US”, “UK”, “Germany”, or “Egypt”.
- A military rank: “General”, “Officer”, or “Civilian”.

Except for the country of origin, these attributes can be modified in time. For instance, one particular user can be upgraded from “civilian” to “officer” or from “secret” to “top secret”. It is infrequent for a military rank to be downgraded but the system should handle this situation for the sake of completeness. For example, a retired military officer working as a contractor could be classified civilian for caveat-evaluation purposes.

4.2. Documents

In this caveat-separation system, users want to access documents. Documents have the following attributes:

- A classification level: “Unclassified”, “Confidential”, “Secret”, or “Top Secret”.
- A caveat. For example: “CEO”, “CANUS”, “CANUSUK”, “NATO”, “UN Coalition”, or “USUK”.
- An affiliation with a project. For example: “Project A”, “Project B”, or “Project C”.

Here, all attributes can be modified during the lifespan of a document, even though certain modifications are unlikely to happen.

4.3. Access Control Tables

There are three access control tables that dictate who can access which documents: the classification table (Table 1), the caveat table (Table 2), and the need-to-know table (Table 3). They are all presented in the following subsections.

4.3.1 Classification Table

Table 1 presents the access relation between the security clearance of users and the classification level of documents. Of course, the higher the classification level of a document, the higher the security clearance required to access it. This
4.3.2 Caveat Table

In Table 2, the caveat of the document dictates acceptable countries of origin for the user who wants to access it. This table is simply an example. Of course, many other caveats and countries exist. However, for the purpose of the current caveat-separation system, it can be considered to be fixed.

4.3.3 Need-to-Know Table

Table 3 determines who, in terms of countries of origin and military ranks, has the “need-to-know” in order to access the documents of a particular project. A user must not only have the proper security clearance and country of origin to be able to access a document but must also have what is called the “need-to-know”. In this caveat-separation system, it is assumed that this “need-to-know” is attributed on a per-project basis and is related to a user’s country of origin and military rank. This table is simply an example and should not be considered fixed once and for all in the caveat-separation system. Indeed, new projects could be created, old projects could be terminated, and entries could be modified or deleted in time.

4.4. Property to Verify

Finally, the following property must be verified to be valid in the design of the current caveat-separation system:

At all times, a user can access a document if and only if the access is granted with respect to the classification, caveat, and need-to-know tables.

This is particularly essential if the attributes and/or the need-to-know table are dynamically modified.

5. Caveat-Separation System: UML Design

For any given design in SOCLE, the designer must include exactly one class diagram, one statechart diagram for each class defined in the class diagram, and one object diagram. In this section, these diagrams are illustrated through the modeling of the case study.

5.1. Class Diagram

Figure 2 presents the class diagram of the simplified caveat-separation system used as a case study. Class Document models a document in the system with its associated security level, caveat, and project. Class User models a user who has access to the system with its associated security clearance, country of origin, and military rank (or civilian). Two roles are represented for users: senders (class Sender) and receivers (class Receiver) of documents. The class Controller manages accesses to documents by users. This class accesses the three tables that dictate which user can access which document. The classes TableSecret, TableCaveat, and TableProject model the classification table (Table 1), the caveat table (Table 2), and the need-to-know table (Table 3), respectively.

In order to be properly handled by the SOCLE tool, the information contained in these three tables is encoded as per tables 4, 5, 6, 7, 8, and 9.

<table>
<thead>
<tr>
<th>Classification level / Security clearance</th>
<th>Enhanced</th>
<th>Confidential</th>
<th>Secret</th>
<th>Top Secret</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified</td>
<td>Granted</td>
<td>Granted</td>
<td>Granted</td>
<td>Granted</td>
</tr>
<tr>
<td>Confidential</td>
<td>Denied</td>
<td>Granted</td>
<td>Granted</td>
<td>Granted</td>
</tr>
<tr>
<td>Secret</td>
<td>Denied</td>
<td>Denied</td>
<td>Granted</td>
<td>Granted</td>
</tr>
<tr>
<td>Top Secret</td>
<td>Denied</td>
<td>Denied</td>
<td>Denied</td>
<td>Granted</td>
</tr>
</tbody>
</table>

Table 1. Classification Table
### Table 2. Caveat Table

<table>
<thead>
<tr>
<th>Project / Country of origin</th>
<th>Canada</th>
<th>United States</th>
<th>United Kingdom</th>
<th>Germany</th>
<th>Egypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>G - O</td>
<td>G - O - C</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>G - O</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>G - O</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>G - O - C</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>G - O - C</td>
<td>G - O</td>
</tr>
</tbody>
</table>

### Legend

- **G**: General
- **O**: Officer
- **C**: Civilian

### Table 3. Need-to-Know Table

<table>
<thead>
<tr>
<th>CEO</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANUS</td>
<td>11</td>
</tr>
<tr>
<td>CANUSUK</td>
<td>12</td>
</tr>
<tr>
<td>NATO</td>
<td>13</td>
</tr>
<tr>
<td>UN Coalition</td>
<td>14</td>
</tr>
<tr>
<td>USUK</td>
<td>15</td>
</tr>
</tbody>
</table>

### Table 6. Caveat Encoding

5.2. Statechart Diagrams

Statechart diagrams are used to define the dynamic behavior of each object in the system. Figures 3 and 4 show the statechart diagrams of the *Sender* and *Controller* classes, respectively. Every class must have its statechart in order for SOClé to be able to predict its dynamic behavior. For the sake of brevity, however, all the statecharts are not presented in this paper.

Notice that the control flow of a statechart is specified by states and transitions. The basic condition for a transition to be fired is that its source state is active. Accordingly, the basic response to firing a transition is the activation of its target state. In the case of composite states, the initial states they encompass are also activated. This basic statechart semantics is adapted from Harel’s statecharts [6].

In addition, transitions are labeled with a trigger, a guard, and a list of actions. Triggers refer to signals (atomic events), method calls, or method returns. For example, the list of actions of a transition labeled with trigger inc (for incrementation) will execute that method’s instructions. Guards are boolean OCL constraints. OCL constraints are presented in Section 6. SOClé supports the following actions: method call/return, field assignment, object creation/deletion, and signal emission.

5.3. Object Diagram

The object diagram is used to define the initial state of all objects in the system. In other words, it gives the initial configuration of the system. Figure 5 shows a possible
Figure 2. UML Class Diagram of the Caveat-Separation System

6. OCL Constraints

The syntax of an OCL constraint $c$ is defined as follows:

$$c ::= v[x] \triangleleft c_1 \triangleleft c_2 | c.f | c_1.\text{iterate}(x_1; x_2 = c_2|c_3)$$

It includes a significant fragment of OCL constraints as defined in [9]. Symbols $v$ and $x$ denote values and variables, respectively. Values include booleans, integers, object names, and lists. Construct $\triangleleft c$ ($c_1\triangleleft c_2$) stands for the application of any of the usual unary (binary) operators on booleans, integers, and lists. Construct $c.f$ returns the value that field $f$ takes in the object $c$. The semantics of these constructs is formally defined as follows:

$$[v]_\rho = v$$
$$[x]_\rho = \rho(x)$$
$$[\triangleleft c]_\rho = \triangleleft [c]_\rho$$
$$[c_1\triangleleft c_2]_\rho = [c_1]_\rho \triangleleft [c_2]_\rho$$
$$[c_1.f]_\rho = \text{heap}([c_1]_\rho, f)$$

The application $\rho(x)$ retrieves the value of variable $x$ in the variable environment $\rho$. The application $\text{heap}([c_1]_\rho, f)$ retrieves the value of field $f$ in the object $c_1$.
fetches the value of field \( f \) according to the ASM function \( \text{heap} \), defined by the UML compiler.

The \textit{iterate} construct is OCL's main collection operator. Informally, it iterates variable \( x_1 \) through the values of the collection denoted by \( c_1 \), stores the successive values of \( c_3 \) in variable \( x_2 \) (which first evaluates to \( c_2 \)), and returns the final value of \( x_2 \). The formal semantics of this construct is simply too complex for the scope of this paper. It is, however, quite expressive and this is why it is used to encode additional collection operators, that are supported by SOCLe. Figure 6 illustrates some of them.

7. Caveat-Separation System: An OCL Constraint

It is possible to give OCL constraints for each statechart of the system in order to validate each module separately but it is even more interesting to define global OCL constraints that validate the entire system as a whole. The OCL constraint defined in Figure 7 is a good example of such constraint. It ensures that the behavior of the controller is correct. Since the security of the whole system depends on the controller, its behavior must be validated. The constraint verifies the three conditions that must be valid prior to the execution of the operation \( \text{access}(u : \text{User}, d : \text{Document}) \):

1. the security clearance of the user is higher or equal to the classification level of the document (1),
2. the nationality of the user is compatible with the caveat of the document (2), and
3. the military rank of the user is compatible with the document project (3) (modeling the need-to-know).

As shown in Figures 8, 9, 10, 11, 12, and 13, SOCLe can automatically verify this constraint for the entire system. This ensures the security of its design.

8. Conclusion and Future Work

This paper has presented a tool that integrates the design of software applications and their security by using OCL constraints on UML diagrams combined with innovative automatic verification techniques. Such a tool is of paramount importance for the design of critical military command and control information systems because these systems have to be not only robust but also secure. Moreover, the detection and correction of errors earlier in the development cycle saves time and money. Finally, a case study on a caveat-separation system has been presented in order to illustrate...
the capabilities of the SOCLE tool.

A few improvements are currently underway. They essentially consist in translating the UML diagrams into an intermediate control flow language before analyzing their behavior and using a local model-checking algorithm to drive the computation of a system’s execution graph. Such improvements will facilitate the implementation of various static analysis techniques such as slicing and code generation. They will also reduce memory requirements as some constraints could be falsified/verified even tough the execution graph is only partially computed.

References

Figure 5. Possible UML Object Diagram for the Caveat-Separation System
context : $Controller :: access(u : User, d : Document)$

pre : $u\.clearance \geq d\.level$
\land
(self.tabC).co \rightarrow \text{filter}\{x|x\.codeP = u\.country\}.cav \rightarrow
\exists y|y\.codeC = d\.caveat$
\land
(self.tabP).cel1 \rightarrow \text{filter}\{x|x\.country.codeCt = u\.country\} \rightarrow
\text{filter}\{y|y\.proj\.codePj = d\.project\}.ra\.codeR \leq u\.rank$

post : true

Figure 7. OCL Constraint for Operation $access(u : User, d : Document)$ of the Controller Class

Figure 8. SCoLE Screenshot: Class Diagram
Figure 9. **SOCLe Screenshot**: Statechart Diagram of the *Controller* Class
Figure 10. SoCle Screenshot: Object Diagram
Figure 11. **SOCLE** Screenshot: Verifying **OCL** Constraint

---

```
== Layer 5: Done ==
== Layer 7: Method Constraints ==
  Extracting Constraints (.xml):
    Class Diagram... Done
  Elaborating Constraints:
  Layer 7: Done ==

== Generating .asm file ==

*** Computing .asm file ***

---

| CRAC's ASM-Compiler & OCL-Model-checker v3 |
| Damien Azambre & Mathieu Bergeron - 2003-2004 |

[1] Opening caveatFinal.asm... done
[2] Creating caveatFinal directory... done
[3] Parsing input file...
  Parsing OCL Expressions... done
  Type-checking OCL Expressions... done
  done in 0.05 s
[4] Computing execution graph...
```

---
Figure 12. Socle Screenshot: Selecting a Node in the Statechart Diagram to Obtain the Corresponding Execution Path
Figure 13. **SOCLE Screenshot: Browsing the Execution Path in the Diagnosis**