Visualization component: Refinement of the requirements

Sequence diagrams and nested call graphs

M. Salois
P. Charland
F. Lemieux
D. Ouellet

Defence R & D Canada – Valcartier

Technical Note

DRDC Valcartier TN 2006-741

January 2007

Author

Martin Salois

Approved by

Yves van Chestein
Head/Information and Knowledge Management

Table of Contents

List of Figures

1 Introduction

In January 2006, DRDC Valcartier started a contract with the University of Victoria, with a subcontract to Thales Canada. A first technical note [1] described the requirements for the contract at a very high level. It contained requirements of a generic nature that the authors believe should be applied to all visualization tools.

This document is an attempt to elaborate on specific aspects of those requirements. It tries to illustrate some typical tasks that architecture recovery and reverse engineers need to perform in their daily routines. It is quite possible that other such documents will be needed along the road to clarify other aspects of the two projects involved in this contract.

The first of these projects is called Opening up Architecture of Software-Intensive Systems (OASIS) [2]. Its objective is to develop technical solutions in order to reduce the time needed to comprehend systems to be integrated into a system of systems. For this project, the sequence diagram is currently the main visualization tool.

The second project shall be known simply as The Other Project.

The choice of using either Zest or SHriMP was difficult. Each tool has its advantages and drawbacks. Since the final decision relies on different factors that cannot all be foreseen at the moment, the decision was made according to short-term practicality. For this reason, Zest [3] has been chosen to implement the sequence diagram and its related ideas for now. SHriMP [4] has been chosen to implement the component for The Other Project.

The next section illustrates the typical tasks mentioned above. The last section gives a lot more detail on how to implement some of these ideas in a sequence diagram.

2 Tasks

This section describes typical tasks that are usual to reverse engineering and architecture recovery. It starts with generic descriptions and ends with two more specific examples.

At the beginning of the OASIS project, an experiment was conducted to try to evaluate the effectiveness of architecture recovery and visualization tools on the comprehension of the programmers [2]. In order to be able to measure the same things for each tool under study, they came up with a series of typical tasks that an analyst would need to perform in a typical architecture recovery scenario. These

tasks give a good idea of the kind of things the current project is trying to achieve through visualization. The relevant tasks can be summarized as:

- Identify components and subcomponents and show the overall structure
- Identify a set of classes relevant to the application domain (e.g. mission, operation, country, and tracks)
- Identify the components involved in
  - interactions with user
  - interactions with the file system
  - interactions with external applications via the network
  - interactions with databases
  - interactions with third-party libraries
  - high-cohesion but low coupling interaction
- Find dependency cycles between components
- Show the overall structure of the application at the component level
- Compute metrics on the program (e.g. afferent and efferent coupling)
- Isolate group of classes according to some criteria
- Instrument the application and run it to:
  - create a call graph to identify the creation/destruction of processes/threads
  - illustrate code coverage
  - find the most solicited areas
  - identify the initialization hierarchy
- Identify the data exchange format (e.g. binaries, XML)

Note that the third task (identifying components) is a subset of the task defined in more detail in the next subsection.

2.1 Pinpoint relevant information

A typical task for a security analyst is to analyze a piece of code to find out if it is secure according to very specific requirements. Another run-of-the-mill task is to analyze the latest vulnerability and try to find out how to patch it and/or how to undo the damage. Other analysts also actively look for new vulnerabilities. Some examples of tasks related to these activities are:

- Is the application:
  - transmitting on the network?
  - managing its temporary file correctly? That is, is it making sure that secure information remains secure and that the files are deleted thoroughly?
  - using a crappy protection mechanism such as the famous Sony rootkit debacle [5]?
  - vulnerable to buffer overflows?
- Verify the extent of a vulnerability and confirm that the patch works. One example for the WMF vulnerability is described in more detail in [6].

Currently, looking for these is largely a manual task. Each analyst has his preferred keywords that he looks for in the programs using simple text search. One obvious way to improve this is by using a nested call graph. The first level displays all the functions of the program. Zooming on a particular function and expanding it shows the inner control-flow graph of the function, in place. Then, color is added to indicate the presence of keywords of interest.

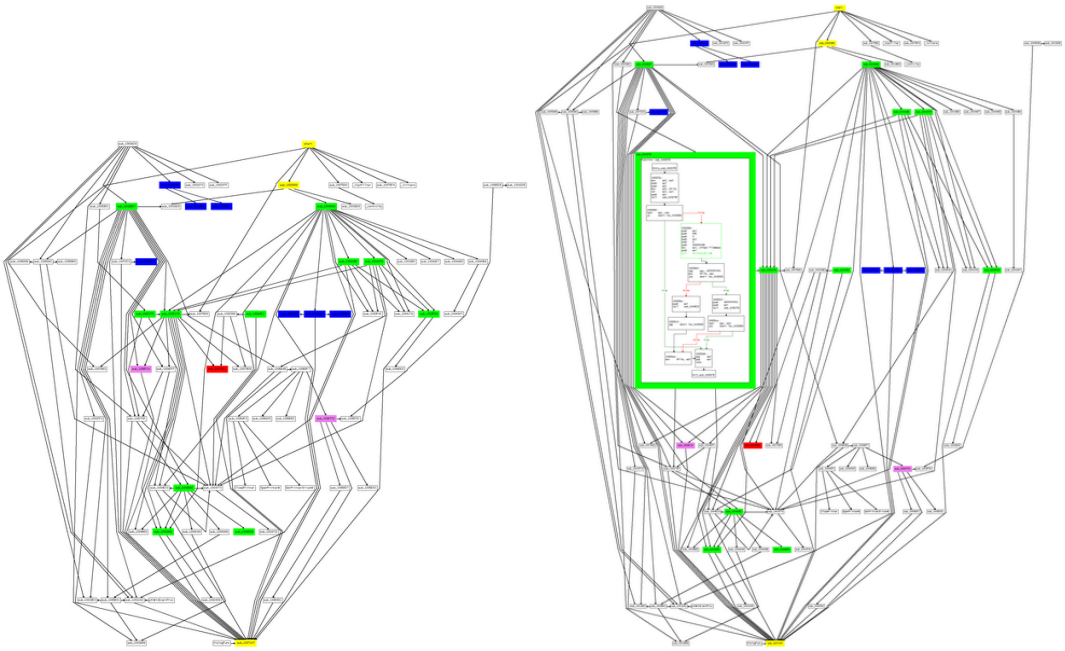
The question remains open as to what type of graph would be best for the outer graph and for the inner graph. One possibility is to use a regular call graph with arrows that are as straight as possible for the outer graph and to use a hierarchical orthogonal layout for the inner graph. An example of this is shown in Figure 1, where green indicates access to the file system. Another possibility for the outer call graph is to use a treemap in which the size of the box depends on the number of children and the order depends on the number of relationships.

This is currently mostly useful for The Other Project but could potentially be quite useful in OASIS.

2.2 Package graphs to document unknown software

Another common problem faced by security analysts is documentation. It is well known that this is not the most exciting part of the job and it is often neglected. One of the reasons why we think using a graphical representation of the code shows promise is that we believe it is easier to manipulate and understand than pure text. However, a side effect could be much better documentation.

Documentation is important in many circumstances but even more so when reporting on software vulnerabilities. The company that receives a report pointing to a vulnerability to their code needs precise information as to where the problem lies. It needs this for three reasons:



(a) The whole nested call graph (b) The same graph with an expanded node

Figure 1: *Nested call graph*

- Quickly verify that the vulnerability is real and exploitable.
- Pinpoint the precise cause of the vulnerability and correct it.
- Validate the patch and make sure that it does not introduce new problems.

Currently, white hats (good hackers) do this using prose. There are many problems with that. As mentioned, they usually do not care too much about this task so they tend to do it rather quickly and dismissively. But the problem is not just there. Writing is difficult and not all analysts can do it clearly. Using the graphical representations that were investigated and utilized by the analyst, it might be possible to come up with an automated way to produce the documentation.

Another use for such a technology could be in architecture recovery; for example, when trying to maintain legacy software or adapt it to a new language. Documentation is often out of date or lacking. The analyst can play with many tools in trying to figure out the inner workings of the program but, eventually, he will need to package all that he has found in a clear document. The same applies to the study of malware (e.g. virus, worm, rootkit).

Therefore, finding a good way to package all the information gathered by analysts in a clear and concise form would be of tremendous help towards sharing information.

This, in turn, would make it easier to move beyond the current cycle of reinventing the wheel that is all too common in the security field (and software engineering!)

This would be really useful for The Other Project.

3 Sequence diagrams

First of all, the authors know that a sequence diagram may not be the best graph representation for program comprehension. However, the information needed to construct such a diagram is readily available and the layout is relatively simple. Furthermore, it is a type of diagram familiar to software developers. Thus, it is believed that it can be useful more rapidly than other types of graph. For these reasons, sequence diagrams are one of the two types of graphs that will be explored in the next phase of the project.

On the other hand, a sequence diagram, in its most basic form, can be too simple for real-life use in the context of program understanding. Also, sequence diagrams were originally designed to represent static information at the design phase. In OASIS, sequence diagrams have been adapted to represent the execution trace of a specific program run. The quantity of information is therefore much greater as calls to method can be repeated over and over again. The display and the capacity of the user can quickly become overloaded if steps are not taken to reduce this information. This is why two major improvements were made to the concept by the OASIS projects:

- The possibility of folding/unfolding classes within subpackages and subpackages within packages. That is, to reduce the information horizontally.
- The possibility of folding/unfolding method calls with upper callers. That is, to reduce the information vertically.

The rest of this section examines two specific problems that could be tackled by further improving the sequence diagram metaphor. It also gives many examples of how these could be achieved. The contractor is responsible for analyzing this information, synthesizing it, determining its feasibility in the time allotted, and proposing the best approach.

3.1 How to compare two traces?

One of the main ideas behind instrumenting a program to record a trace of its execution via method calls was the fact that this applies a first filter of sort.

Let's say an analyst is trying to understand a specific functionality, say how a file is saved to disk. One way to do this is to try to determine statically which part of the program is involved in this process. Another way is to set up a trace of the execution, to fire up the program, then to save a file to disk within the program and exit. Now, it is certain that somewhere within the complete trace, the sequence of method calls that leads to saving a file on disk is recorded. The problem now is to find where.

A possibly elegant solution is to record two traces of execution. In the first one, the analyst simply opens and closes the program. In the other, the analyst saves a file to disk. Now, in theory, the difference between the two traces contains the area of interest that pertains to saving a file on disk. The problem now is to find the best way to illustrate the differences between the two traces. This is where a good visualization comes to the rescue.

Here, one can draws from concurrent versioning systems, such as CVS and Subversion, that are quite good at recording the differences between two versions of a source code file. In practice, there is always the problem of determining which of the two versions is the reference. Comparing A to B in this case is different from comparing B to A since differences will be annotated differently. Putting that aside, there are four things that can happen to an element of comparison:

Identical The element is exactly the same in both A and B.

Deleted The element is present in A but absent in B.

Added The element is absent in A but present in B.

Modified The element is present in both A and B but with different parameters.

The challenge in this context is to find the best way to highlight these changes in a sequence diagram.

The most obvious solution, and the one used in most text comparison, is to use colors. For example, identical elements remain unchanged while deleted element are in red, added elements are in blue, and modified element are in green. As this is probably very limited on large execution traces (it does not scale, hence the next subsection!), finding an effective way of displaying these changes is an interesting research problem.

3.2 How to scale?

Although folding and unfolding provides a very good first step towards managing a large quantity of information in a sequence diagram, it is not sufficient. In this

subsection, ideas are proposed to take this to the next level. This can also be an interesting research problem by itself.

3.2.1 Automated customization

The idea is that there are certain things that are probably not of interest at first glance or are superfluous towards a better comprehension. Also, the more stuff there is on the screen, the less detail are required for each item.

For example, when trying to understand a whole system, an analyst is probably interested in first seeing an overview of the whole system. The first view might therefore show only the high-level packages. Determining which of the packages is high-level automatically is an open question...

Another idea is to adjust the level of detail according to what can currently be seen on the screen. For example, if the text of a label does not fit in a box or between two call lines, then do not display it.

3.2.2 Filtering

A relatively simple improvement that could help greatly in getting a basic understanding of a large system is intelligent and automated filtering.

In a dynamic trace, there are many repetitive elements that are not necessarily useful for comprehension. For example, in Java, each time an object is called for the first time, its `init` function gets called and stuff gets put on the stack and so on. This is information that the programmer already knows and it occupies a lot of space. Imagine a simple trace with 100 hundreds objects. That makes 100 hundred calls to `init` and, thus, 100 calls that do not contribute to comprehension. In a sequence diagram, since the return value is also indicated, that makes 100 arrow-method-arrow blocks less to display.

Other candidates for automatic filtering are get/set methods and standard library calls (e.g. `toLowerCase`, `addXListener`, etc.) Obviously, this information may be useful in some context so filters should be highly customizable. This also confirms the usefulness of another proposed improvement: 3.2.1 Automated customization.

3.2.3 Focus

This feature is already implemented in the Zest prototype but it may be possible to improve on it. The idea is to highlight the current focus of attention and dim the rest of the view or make it disappear entirely. The goal, once again, is to limit

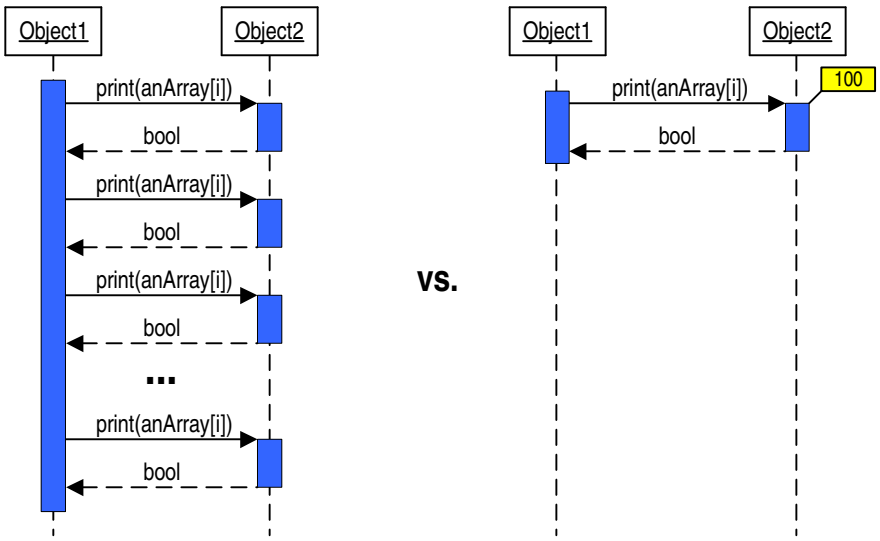


Figure 2: *Grouping loops*

the cognitive load on the analyst while helping him keep his mental picture of the whole system.

The reordering of lifelines depending on the focus might be an interesting concept. Still, it is not clear if this would be more confusing than anything else so it should be kept as a side idea for further experimentation.

3.2.4 Grouping of program structures

The problem with a dynamic trace is that all actions are explicit and given one after the other. Let's say there is a loop to print to screen all the values of a 100-element array. That creates 100 hundred arrow-method-arrow blocks to display, one after the other. It would greatly declutter the display to find a way to illustrate this in just one loop. A possible solution is shown in Figure 2.

Once again, it is conceivable that an analyst would require expanding this group. Maybe there could be a little '+' sign next to the '100'.

3.2.5 Hierarchical header

When folding and unfolding packages and subpackages, it is easy for the analyst to get lost and not remember what's in what. In order to clarify this and provide yet another way of looking at the information, it could be useful to conceive a header for this. Also, when scrolling down a large sequence diagram, it is easy to forget which lifeline belongs to which object. This could also be part of a header that is

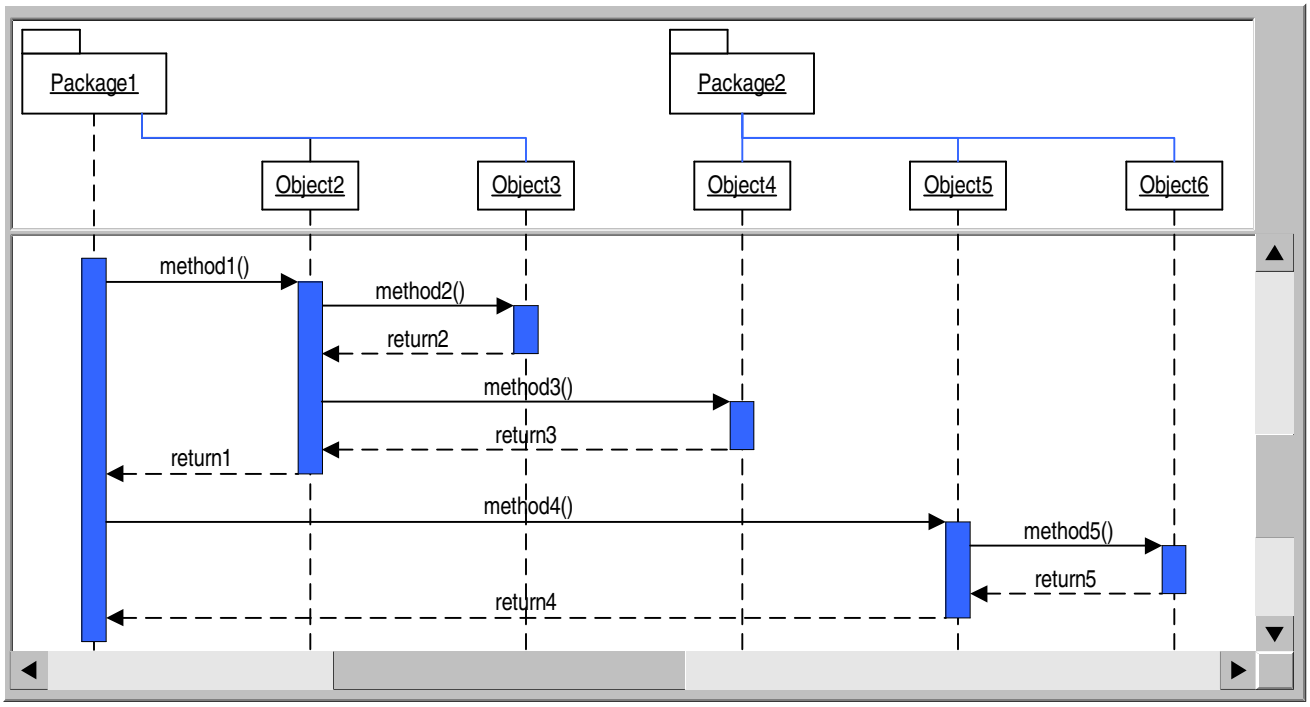


Figure 3: *A header that shows the hierarchy and stays at the top of the display*

permanently displayed at the top of the view. For those familiar with Microsoft Excel, this would be similar to the “freeze pane” function.

Different options could be tried to display the hierarchy. One example is provided in Figure 3. Feel free to experiment.

3.2.6 Multiple views

It might be a good idea to give the analyst the possibility of opening multiple synchronized views of the sequence diagram.

In many cases, the end point of a method call will be outside the screen as zooming is required to be able to read the labels and see what is going on. Two views might be good in this particular instance. The left view shows the start point of the method call, with its surrounding area of interest; the right view shows the end point, also with its surrounding area of interest. An example is shown in Figure 4.

Finding the best way to navigate these views can be a small research goal. So is the question whether or not more than two views would be helpful.

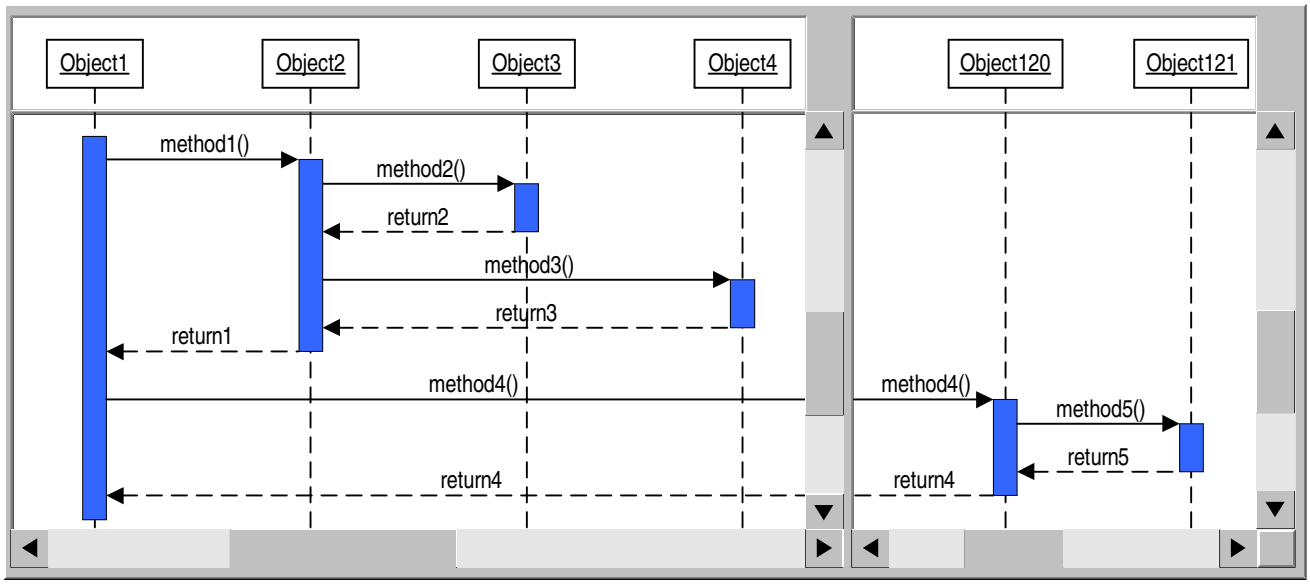


Figure 4: *Two views: the right column displays relevant but out-of-screen lifelines*

3.2.7 Saving views

This element is very simple to explain. It consists in adding the concepts of TagSEA [7] to the sequence diagram. The implementation, however, might not be as simple and may pose an interesting research problem.

References

1. Charland, Philippe, Lemieux, François, Ouellet, David, and Salois, Martin (2006). Basic Requirements for a Visualization Component: Kick Off Meeting Notes. (Technical Note TN 2006-064). Defence Research Establishment – Valcartier. Val-Bélair, Québec, Canada. Read the PDF.

2. Charland, Philippe, Dany, Dessureault, Lizotte, Michel, Ouellet, David, and Nécaille, Christophe (2006). Using software analysis tools to understand military applications. (Technical Memorandum TM 2005-425). DRDC Valcartier. Québec (QC) Canada.

3. The Chisel Group (2006). Zest: The Eclipse Visualization Toolkit. http://www.thechiselgroup.org/zest.

4. The Chisel Group (2006). SHriMP. http://www.thechiselgroup.org/shrimp.

5. Wikipedia (2005). 2005 Sony BMG CD copy protection scandal. http://en.wikipedia.org/wiki/2005_Sony_CD_copy_protection_scandal.

6. Lavoie, Yvan and Salois, Martin (2006). Windows Metafile Vulnerability: A Small Case Study. (Technical note TN 2006-106). DRDC Valcartier. Québec (QC) Canada.

7. The Chisel Group (2006). TagSEA. http://tagsea.sourceforge.net.

List of Acronyms

CVS Concurrent Versioning System

DRDC Defence Research & Development Canada

OASIS Opening up Architecture of Software-Intensive Systems

SHriMP Simple Hierarchical Multi-Perspective

WMF Windows Metafile

XML Extensible Markup Language

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R & D Canada – Valcartier
2459 Pie-XI Blvd North, Québec, QC, Canada

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title).

Visualization component: Refinement of the requirements

4. AUTHORS
(Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

Salois, M. ; Charland, P. ; Lemieux, F. ; Ouellet, D.

5. DATE OF PUBLICATION (month and year of publication of document)

January 2007

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc).

16

6b. NO. OF REFS (total cited in document)

7

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered).

Technical Note

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address).

Defence R & D Canada – Valcartier
2459 Pie-XI Blvd North, Québec, QC, Canada

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Specify whether project or grant).

15AV20

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written).

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique.)

DRDC Valcartier TN 2006-741

10b. OTHER DOCUMENT NOs. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Defence departments and defence contractors; further distribution only as approved
( ) Defence departments and Canadian defence contractors; further distribution only as approved
( ) Government departments and agencies; further distribution only as approved
( ) Defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution beyond the audience specified in (11) is possible, a wider announcement audience may be selected).

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

visualization, program comprehension