Tool-Supported Step-By-Step Debugging
for the Object Constraint Language

Lars Schütze, Claas Wilke, and Birgit Demuth

Technische Universität Dresden
Software Technology Group, Dresden, Germany
{lars.schuetze|claas.wilke|birgit.demuth}@tu-dresden.de

Abstract. Although originally designed as an extension for the Unified Modeling Language (UML), the Object Constraint Language (OCL) has been broadly adopted in the context of UML as well as other modeling and domain-specific languages. However, appropriate tooling, supporting software developers on using OCL is still scarce and lacks debugging support. As OCL constraints are likely to become rather complex for real world examples, it is typically hard to comprehend the influence of single OCL expressions on the result of an evaluated OCL constraint. Therefore, debugging is of topmost importance for both constraint comprehension and maintenance. In this paper, we evaluate existing debugging tools for OCL and come to the conclusion that no real step-by-step debugger for OCL exist, yet. Therefore, we analyze requirements for OCL debuggers and present an OCL debugger implementation for Dresden OCL.

Keywords: OCL, Debugging, Tracing, IDE4OCL.

1 Introduction

Within the last 20 years, the Object Constraint Language (OCL) [1] has become a widely adopted constraint language used in the context of the Unified Modeling Language (UML) [2] as well as transformation languages like Query View Transformation (QVT) [3], and other domain-specific modeling languages [4]. However, for the further acceptance of OCL in research and industrial application, adequate tooling for an *IDE4OCL* is still missing [5]. An online survey conducted by Chimiak-Opoka et al. among more than 100 OCL users in 2009 and 2010 revealed, that the most-wanted features for such integrated development environments (IDEs) comprise adequate OCL debugging and refactoring support [6].

Whereas refactoring focuses on the maintenance and optimization of OCL expressions, debugging allows for their systematic execution, and therefore, the identification of implementation faults within OCL constraints [5]. Although OCL refactoring has been investigated and implemented for existing tooling recently [7,8], accurate support for OCL debugging is still an open task. First approaches support OCL evaluation tracing [9], but—to the best of our knowledge— no real step-by-step debugging for OCL exist, yet. Thus, in this paper, we present

a step-by-step debugger for OCL, implemented for an open-source OCL tool, namely Dresden OCL[1]. We identify requirements for OCL debugging and evaluate existing OCL tools w.r.t. their support for OCL debugging. Afterwards, we present our implementation of a step-by-step debugger for Dresden OCL.

The remainder of this paper is structured as follows. In Sect. 2 we define the term *OCL debugging* as understood in this paper. Afterwards, in Sect. 3 we identify requirements and useful features for step-by-step OCL debuggers. Following, we investigate OCL and OCL-related tools in Sect. 4 w.r.t. the identified requirements for OCL debugging functionality. Finally, Sect. 5 outlines our implementation for OCL debugging and Sect. 6 concludes this paper.

2   Background

The use of *debugging* has a long tradition in electronics and computer science. Thus, various definitions and understandings of the term debugging exist. Since the 1960s, debugging is recognized as a process of finding and correcting *bugs* (errors, defects) in computer programs. However, debugging is often explained as the process of correcting syntactic and logical errors, detected during coding only [10]. Within the same meaning, Sommerville emphasizes in his software engineering text book [11] that debugging and *defect testing* are different processes: defect testing establishes the existence of defects; debugging is concerned with locating and correcting these defects. A state-of-the-art debugging tool, mostly called a *debugger*, generally provides both functionalities. It is used to test and debug programs [12], and offers functions such as

- *Step-by-step execution* of a program one *line* at a time: therewith the programmer may examine the state of related data, before and after execution of a particular line of code,
- *Breaking* or stopping the program to examine the current state at a specified instruction by means of a *breakpoint*,
- *Tracing* the values of variables,
- *Modifying* the program state while it is running,
- *Logging* of debugging activities,
- *(Semi-)Automating* test case generation out of logged debugging results.

Although some of these debugging aspects can be considered as being highly related to imperative languages (e.g., step-by-step execution of a program executing one line at a time), similar functionality is also required for declarative and side-effect free languages such as the OCL. Although each OCL constraint or query may be expressed as one single line of code, this line of code can comprise many tree-structured OCL expressions which are evaluated during interpretation in a visitor-based manner. Thus, for OCL programmers it is helpful in analyzing bugs to know the immediate results of each subsequent constituent of an OCL expression. Therefore, we test and debug individual OCL expressions instead

---

[1] http://www.dresden-ocl.org/

```
1 context Person:
2 inv: getChildren()->size() >= 0
```
Listing 1. An example OCL constraint.

```
1 context Person def: getAge() : Integer =
2 self.age + (if self.hadBirthday then 1 else 0 endif)
```
Listing 2. A constraint with an if expression.

of imperative program statements that are typically coded by one statement per (code) line. Besides step-by-step execution of OCL sub-expressions, OCL programmers are interested in tracing the values of variables as well as using conditional breakpoints in an expression to examine their current state. We call this technique *step-by-step debugging*.

3   Requirements

In the following, we discuss requirements and useful features for step-by-step OCL debuggers. The requirements emerge from the debugging features described by Zeller [12] outlined above, as well as from an analysis of the functionality provided by existing OCL debuggers such as USE [9] and debug tools for imperative programming languages such as the Eclipse Java development tools (JDT)[2].

/R1/ Step-by-step debugging A step-by-step OCL debugger should support step-wise debugging of OCL expressions. The debugger should suspend its OCL evaluation when reaching a breakpoint defined on an OCL expression. Such a breakpoint could be either defined on the respective OCL expression in an OCL editor (concrete syntax), or by marking the respective OCL expression in a tree-based presentation of nested OCL expressions (abstract syntax).

/R1.1/ Highlighting the currently interpreted OCL expression As OCL is a declarative language, a single line of OCL code in concrete syntax is likely to contain several OCL expressions (e.g., the OCL constraint given in Listing 1 consists of three operation call expressions, one integer literal expression and one implicit variable call expression referring to the self variable). Thus, if an OCL debugger is suspended during interpretation, it should highlight the currently evaluated OCL expression within an OCL editor, or within a tree-base representation of its abstract syntax, or optimally, both. For the given example, either one operation call, the integer literal or the whole constraint (at the beginning or the end of the constraint's interpretation) should be highlighted.

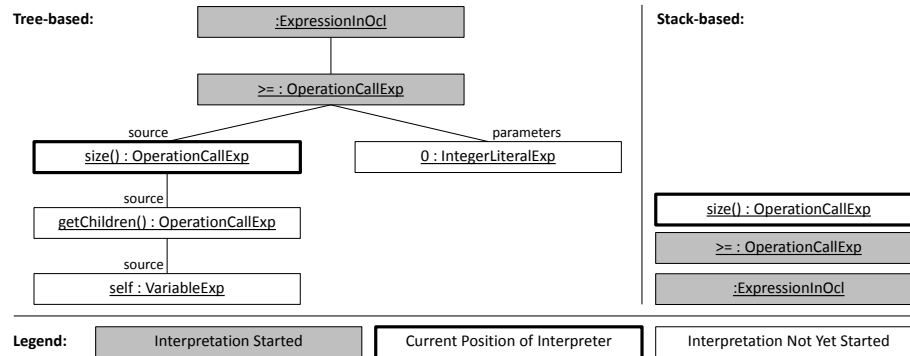
[2] http://projects.eclipse.org/projects/eclipse.jdt

Fig. 1. Tree- and stack-based representation of an OCL constraint during debugging.

/R1.2/ Stepping support Existing debuggers (e.g., debuggers based on the Eclipse debugging framework) support different kinds of stepping for step-by-step debugging; namely *step into*, *step over* and *step return*. OCL debuggers should support these three kinds of debugging steps. For example, for the if expression shown in line 2 of Listing 2 and the current position of the interpretation being after the evaluation of the *condition* `self.hadBirthday` to `true`, a *step into* would continue the evaluation and suspend again in front of the **then** expression `1`, whereas a *step over* and a *step return* would both result in a suspension in front of the the + operation.

/R2/ Visualization of the visited OCL expressions During OCL interpretation, nested OCL expressions are evaluated in a visitor-like manner. To visualize the currently evaluated OCL expressions, OCL debuggers should visualize the visited OCL expressions either in a tree-based or stack-based manner. For example, for the OCL invariant shown in Listing 1 and the interpretation being suspended at the `size()` operation call, the tree- and stack-based representations of the currently evaluated OCL constraint are shown in Fig. 1.

/R3/ Visualization of results for evaluated OCL expressions Although the visualization of the currently evaluated OCL expressions in a tree- or stack-based manner can ease the understanding of OCL interpretation, an OCL debugger should visualize the results of already evaluated nested OCL expressions, giving an overview on all evaluated expressions and their results, and thus, the causes for the sub-expressions being evaluated in the following (e.g., for an if expression this visualization would show the evaluated condition as well as whether the **then** or the **else** branch have been evaluated). If a tree-based visualization for the currently evaluated OCL expression is used (/R2/), the same visualization can be used by annotating the nodes in the tree with their evaluation results.

/R3.1/ Filtering of evaluated sub-expressions Although the tree-based visualization may be helpful to further understand the evaluation of an OCL expression, it can also contain too much information in some scenarios. Therefore, OCL debuggers can provide support filtering the expression trace (e.g., sub-expressions causing the final result to be `false` instead of `true` could be shown exclusively to detect bugs in invariants) [9].

/R3.2/ Visualization of unnecessary sub-expressions Similar to other interpreters, OCL interpreters can be optimized to skip the evaluation of sub-expressions having no impact onto the final result of a nested OCL expression (e.g., for the evaluation of an expression `(a or b)`, `b` has not to be evaluated, if `a` results in `true`). Further examples are iterators whose interpretation can be cancelled if one visited element fulfills the iterator's condition (e.g., for the `any` iterator). However, it can be helpful to fully interpret such OCL expressions and to present their results in the tree-based evaluation trace, as the evaluated expressions letting their evaluation being superfluous may contain bugs as well [9].

/R4/ Tracing of variable values During debugging, the values of currently visible and accessible variables should be presented, to ease the understanding of the current program state. For OCL debuggers, all variables defined in `let` expressions as well as possible parameters emerging from the constraint's context (e.g., when interpreting a `pre` or `post` constraint of an operation having arguments) should be visualized. Besides, already evaluated expressions should be visualized as well. For example, consider the interpretation of the `if` expression shown in line 2 of Listing 2, being suspended after interpreting the *condition* of the `if` expression. At this point in time, the evaluated value of the condition should be visualized as well to ease the understanding for the user why either the interpretation of the `then` expression, or the `else` expression will follow. Furthermore, the the `self` variable for the currently interpreted constraint should be visualized; as well as further predefined variables (e.g., the `result` variable, if a `post` constraint is interpreted).

/R5/ Conditional breakpoints For existing debuggers (e.g., the Java debugger of the JDT), it is very common to provide conditional breakpoints. Thus, breakpoints can be annotated with Boolean conditions, specifying whether or not the debugger should suspend, when the breakpoint is reached. For OCL debuggers, these conditions should support references to variables, operations and properties visible at the OCL expression(s) corresponding to the breakpoint.

/R6/ Watch expressions Besides conditional breakpoints, existing debuggers support *watch expressions*. Watch expressions are additional expressions being evaluated for the current context of an interpretation. For example, if the debugger is suspended during the interpretation of a constraint defined on a `Person` class, a watch expression could contain a call to an operation of this

	/R1/	/R1.1/	/R1.2/	/R2/	/R3/	/R3.1/	/R3.2/	/R4/	/R5/	/R6/	/R7/
USE	-	-	-	-	✓	✓	✓	-	-	-	-
SQUAM OCL	-	-	-	-	✓	-	-	-	-	-	-
QVT-O	(✓)	-	-	✓	-	-	-	✓	✓	✓	-
MDT/OCL	-	-	-	-	-	-	-	-	-	-	-
Dresden OCL	✓	✓	✓	✓	✓	(✓)	-	✓	-	-	-

Table 1. OCL (related-)tools and the requirements they support for OCL debugging.

class (e.g., `getAge()`). The expression could help the user to further evaluate the current state of the objects being interpreted, besides the variables visible in the variables view.

/R7/ Program and program state modification As introduced in Sect. 2, debugging can include the modification of the debugged program, while it is running. For example, the JDT Java debugger allows hot code replacement during debugging. In the context of OCL, several different modifications while debugging an OCL constraint are possible. First, the debugged constraint itself can be modified to fix bugs while debugging the constraint in a certain context (e.g., a `false` Boolean literal could be fixed to be `true` instead). Second, besides the constraint, the model, the constraint is referring to can be modified (e.g., an operation's semantics given in the model could be modified during debugging). Third, the state of the model instance, in which context the constraint is currently debugged, can be modified (e.g., the `age`) of a `Person` being the context of a constraint could be modified). Finally, the modification of visualized variables' values or the evaluation results of individual OCL expressions in the tree-based evaluation view could be supported, to check the direct impact of other values during debugging.

4 Related Work

Besides Dresden OCL, other OCL and OCL-related tools exist (e.g., the UML-based Specification Environment (USE), the Systematic Quality Assessment of Models OCL Tool (SQUAM OCL), Eclipse MDT/OCL, and QVT-O). In the following, we investigate these tools regarding their support for OCL debugging, and name the requirements supported by our current implementation of an OCL debugger for Dresden OCL. A summary of the requirements supported by all tools discussed below is given in Table 1.

USE[3] offers an *evaluation browser* visualizing the evaluation of OCL expressions as a tree [9]. The evaluation browser can be considered as an OCL debugger (without support for step-wise OCL debugging), supporting the requirement /R3/. Besides, the evaluation browser can be filtered (e.g., to show

---

[3] http://sourceforge.net/projects/useocl/

failed evaluations only and to examine the root cause of the failure or unexpected result), supporting the requirements /R3.1/ and /R3.2/.

SQUAM OCL[4] offers the ability to trace the evaluation of selected OCL expressions on a single model element. The trace is presented as text output on the console listing all evaluation steps and their results. Therefore, SQUAM OCL OCL can be considered as supporting the requirement /R3/ in a text-based manner.

Although the QVT-O tools of the Eclipse modeling tools[5] do not support OCL debugging, they support debugging of QVT-O expressions, which can be considered as rather similar functionality, as QVT-O includes OCL expressions. The QVT-O debugger is integrated into the Eclipse environment and offers a large subset of debugging functionality for QVT-O such as stepping /R1/, a variables view /R4/, conditional breakpoints /R5/, and watch expressions /R6/. However, although the QVT-O debugger supports line-based stepping, stepping at the OCL expression level (/R1.1/ and /R1.2/) is not supported, as stepping is only possible for individual lines of QVT-O code in the QVT-O editor.

Another widely used OCL tool is MDT/OCL provided by the Eclipse modeling tools[6]. MDT/OCL currently does not offer interactive debugging support. At best, its OCL Console can be used to evaluate partial OCL expressions. However, OCL debugging is planned to be realized for subsequent releases[7].

Following, we shortly name the requirements supporty by our implementation of an OCL debugger for Dresden OCL presented in the subsequent section. Our solution supports step-by-step debugging /R1/ (including its sub-requirements and the visualization of visited OCL expressions /R2/. Besides, variable tracing /R4/ and expression tracing /R3/ are supported. The sub-requirement for filtering /R3.1/ is partly supported, as expressions can be filtered w.r.t. `true` and `false` values. The evaluation of unnecessary OCL expressions /R3.2/ is not supported. Conditional breakpoints /R5/, watch expressions /R6/ and program state modification /R7/ are targets for enhancements in future works.

Apart from debuggers for OCL, debuggers for other declarative languages exist. Cabellero et al. present a debugging framework for SQL views [13]. Their tool visualizes the computation of an SQL view as a computation tree (similar to a tree-based representation of OCL expressions), allowing users to inspect the computation results of individual nodes within the computation tree and mark their results as being either correct or incorrect to detect sub statements or nested views causing the erroneous result of an SQL view. A more advanced SQL debugging tool is proposed by Herschel et al. [14]. Their Eclipse-based tool allows for inspection why specific tuples do not appear within a result set by computing the missing relations for the expected results.

---

[4] http://squam.info/?p=142
[5] http://www.eclipse.org/mmt/?project=qvto
[6] http://www.eclipse.org/modeling/mdt/?project=ocl
[7] http://wiki.eclipse.org/MDT/OCL/Debugger

5   Implementation

As outlined above, based on the identified requirements for OCL debuggers, we built a ready-to-use OCL debugger for Dresden OCL [15]. Although tree-based debuggers such as the USE debugger can help to visualize the interpretation results of OCL expressions, we argue that for real debugging, step-by-step debugging including the highlighting of OCL expressions in a textual OCL editor is required. Especially for the debugging of more complex OCL statements (e.g., iterator or even nested iterator expressions), a simple tree-based presentation is insufficient, as the text editor integration further increases the understanding of evaluation results for individual nested OCL expressions.

The existing OCL interpreter of Dresden OCL was extended by an OCL debugger that suspends on breakpoints and propagates variable traces to the Eclipse UI. To provide native look and feel for Eclipse-integrated debugging, the OCL debugger was realized with the Eclipse debugging framework and was fully integrated with the EMFText-based OCL editor and parser of Dresden OCL [16]. During debugging, breakpoints can be defined in the OCL editor and respective expressions are highlighted when stopping on a breakpoints during debugging. Currently, our OCL debugger supports step-by-step debugging /R1/, the visualization of currently interpreted OCL expressions as an expression stack /R2/, as well as viewing the debugged expression trace /R3/ and variable tracing /R4/. Support for conditional breakpoints /R5/, watch expressions /R6/, and program state modification /R7/ are planned features for future works.

Figure 2 shows a screenshot of the debugger in the Eclipse debug perspective, while debugging a constraint from the Royal & Loyal example by Warmer and Kleppe [17]. At the top left, the expression stack of the currently debugged constraint is visible. Besides, on the right, the variables view illustrates the currently visible variables. At the currently reached breakpoint, after the evaluation of the property call `levels`, the `self` variable is the only visible variable. However, the variables view also contains the source and the result of the interpreted expression call named as `oclSource` and `oclPropertyValue` and the result of the currently interpreted constraint `oclResult`, which is an empty **Set** by now but may by filled during the interpretation of the **select** and **collect** iterators yet to be debugged and interpreted. Below, in the center, the OCL editor visualizes the currently debugged constraints and OCL expressions. Below, at the bottom of the screen, a tracer view illustrates the expression trace of yet debugged OCL expressions and their results in a tree-like manner.

6   Conclusion

In this paper, we identified requirements for OCL debugging and evaluated existing OCL tools w.r.t. debugging support. We showed that none of the investigated tools fully supports step-by-step debugging for OCL, yet. Although, OCL debugging has been highlighted as one of the most-wanted features in related work [6]. Therefore, we presented our solution for a step-by-step OCL debugger that is
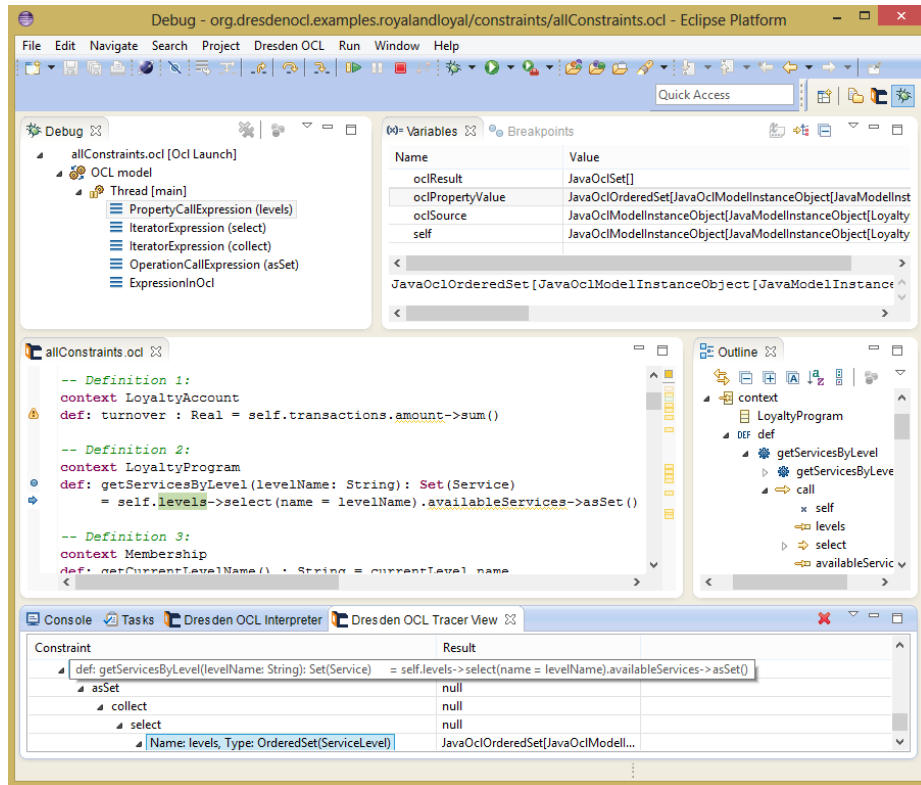
Fig. 2. The OCL debugger, while debugging the Royal & Loyal example from [17].

based on the Eclipse debugging framework and extends the existing interpreter of Dresden OCL. The OCL debugger has been extensively tested and is publicly available[8]. To the best of our knowledge, this is the first tool supporting step-by-step debugging for OCL. In future works we plan to address the missing features discussed in Sect. 3 such as conditional breakpoints, watch expressions and program state modification. With our step-by-step OCL debugger, OCL tooling is another step further on its way towards an IDE4OCL as envisioned by Chimiak-Opoka et al. [5].

Acknowledgements

---

[8] http://www.dresden-ocl.org/

References

1. Object Management Group (OMG): Object Constraint Language. Version 2.3.1. Online available specification (January 2012)
2. Object Management Group (OMG): Unified Modeling Language. Version 2.4.1. Online available specification (August 2011)
3. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation, Version 1.1. Online available specification (January 2011)
4. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Integrating OCL and Textual Modelling Languages. In: Models in Software Engineering. Volume 6627 of Lecture Notes in Computer Science., Berlin / Heidelberg, Springer (2011) 349–363
5. Chimiak-Opoka, J.D., Demuth, B., Silingas, D., Rouquette, N.F.: Requirements analysis for an integrated OCL development environment. Electronic Communications of the EASST **24** (2010)
6. Chimiak-Opoka, J., Demuth, B., Awenius, A., Chiorean, D., Gabel, S., Hamann, L., Willink, E.: OCL Tools Report based on the IDE4OCL Feature Model. Electronic Communications of the EASST **44** (2011)
7. Correa, A., Werner, C.: Refactoring object constraint language specifications. Software & Systems Modeling **6**(2) (2007) 113–138
8. Reimann, J., Wilke, C., Demuth, B., Muck, M., Aßmann, U.: Tool Supported OCL Refactoring Catalogue. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, New York, ACM (2012) 7–12
9. Brüning, J., Gogolla, M., Hamann, L., Kuhlmann, M.: Evaluating and Debugging OCL Expressions in UML Models. In: Test and Proofs. Volume 7305 of Lecture Notes in Computer Science., Berlin / Heidelberg, Springer (2012) 156–162
10. Adrion, W.R., Branstad, M.A., Cherniavsky, J.C.: Validation, verification, and testing of computer software. ACM Computing Surveys **14**(2) (June 1982) 159–192
11. Sommerville, I.: Software Engineering. 8th edn. Addison Wesley, Boston, MA (2007)
12. Zeller, A.: Why programs fail - a guide to systematic debugging. Elsevier (2006)
13. Caballero, R., Garcí-Ruiz, Y., Sáąenz-Pérez, F.: Algorithmic Debugging of SQL Views. In: Perspectives of Systems Informatics. Volume 7162 of Lecture Notes in Computer Science., Berlin / Heidelberg, Springer (2012) 77–85
14. Herschel, M., Hernández, M.A., Tan, W.C.: Artemis: A System for Analyzing Missing Answers. In: 35th International Conference on Very Large Data Bases (VLDB'09), New York, ACM (2009)
15. Schütze, L.: OCL Debugging for Dresden OCL. Bachelor's thesis, Technische Universität Dresden (2013)
16. Aßmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowinski, J., Reimann, J., et al.: DropsBox: the Dresden Open Software Toolbox. Software & Systems Modeling **11** (2012) 1–37
17. Warmer, J., Kleppe, A.: The Object Constraint Language - Getting Your Models Ready for MDA. 2nd edn. Pearson Education Inc., Boston, MA (2003)