

Defining and Qualifying Components in the Design Phase

Andrew O'Fallon, Orest Pilskalns, Andrew Knight, Anneliese Andrews
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164
{ aofallon, orest, aknight, aandrews }@eecs.wsu.edu

Abstract

Component based development in the design phase necessitates a comprehensive look at both static and dynamic UML views. If a design is to incorporate third-party components, one must define component interfaces. We propose a method for defining components in the design phase that can be used for qualification purposes. Coupling and frequency metrics are used to make component definition decisions. Component interface definitions allow for qualifying candidate components.

1. Introduction

Software design is increasingly including component based software development [3]. This happens from two perspectives: (1) a software designer wants to use components and needs to define how components fit with the remainder of the design. Then candidate components must be evaluated whether they fit into the design. The latter is part of component qualification. (2) a software designer wants to design components for reuse. This could be as part of a product line architecture, or as part of a set of components developed for reuse.

In either case, one needs to determine what the component boundaries and interfaces are. Second, candidate components need to be evaluated how well they fit into the overall design. i. e. how well they fit the component boundaries.

We propose an analysis method that works for designs expressed in UML Class Diagrams and Sequence Diagrams. It is derived from Pilskalns et al. [11]. In [11], a model is derived from Class Diagrams and Sequence Diagrams that integrates both structural and behavioral characteristics of the design. It is used to generate and execute tests. This paper uses the same model to evaluate component boundaries and define components with high cohesion and low coupling. Candidate components are then qualified by how well they fit the interfaces of the component with the rest of the design.

Section 2 describes existing work on component selection and qualification. Section 3 explains the approach used

for component definition. Section 4 defines the method for component qualification. Section 5 illustrates the analysis method on an example. Section 6 draws conclusions and suggests further work.

2. Background

In [5, 6] Kontio et al. apply a selection and evaluation method to multiple case studies. The method investigated is referred to as OTSO (Off-The-Shelf Option). OTSO describes a systematic approach to selecting packaged components. The method includes six phases: search, screening, evaluation, analysis, deployment, and assessment. Lester et al. [8], apply the idea of using stereotypes, class compartments, and association rules for qualifying the reuse of software artifacts. These UML constructs are used to define search criteria for reuse candidates. The stereotype is used to limit the search of objects to those objects that contain the stereotype or are derived from the object with the stereotype. Attribute-Value classification can be used to provide a structured way to integrate association roles into the search criteria of an object.

The COTE (Component TEsting) project [4] is concerned with developing an integrated environment (IE) for qualifying and testing components. The research is primarily interested in using the IE for components modeled in UML. Sequence diagrams are used to generate UML test profiles.

These methods are more concerned with component evaluation and qualification than definition. Further, they are fairly high level. We see our method as a more detailed analysis approach whose results can be used in the context of an OTSO evaluation. Similarly, our component qualification method can be seen as a method for selecting candidate components that can then be tested using COTE.

3. Defining Components

We define components by (1) creating a model that merges the static and dynamic information of UML Class

and Sequence diagrams, (2) applying an operation profile to the model to collect metrics, and (3) analyzing the metrics to identify boundaries in the model for defining potential component interfaces. The first step in defining components is to convert the class diagrams and sequence diagrams into a directed acyclic graph that can be analyzed for cohesion [1] and coupling (Constrained Object Method Directed Acyclic Graph or COMDAG). We do this in two steps based on Pilskalns et al. [11]. First we convert classes into constrained class tuples (CCTs) that describe attributes, methods, and inheritance relationships of a class. A CCT is contained in each node of the COMDAG and represents an instantiated class. A set of CCTs can be used to define a component, since all of the interface information is available. The sequence diagrams are converted into graphs (COMDAG), starting with the first method call, following the paths through the sequence diagram. Nodes are defined by the method, object, and classes involved. Edges connect method sequences as specified in the sequence diagram. Table 1 shows the definition of the CCT as specified in [11]. Here we do not need all parts of this definition, since we do not need to generate and execute test cases.

The COMDAG can be constructed by mapping elements of the Sequence Diagrams to a graph. The COMDAG is a tuple $\langle V, E, s \rangle$ where V is a set of vertices, E is the set of edges, and s is the starting vertex. Each vertex, v , is defined by the triple $v = \langle o, \langle M \rangle, CCT(c) \rangle$, where o is an object, $\langle M \rangle$ is a method tuple, c is a class. An edge E , represented by the tuple $\langle v_1, v_2 \rangle$, consists of a pair of vertices that represent the ordering between vertices v_1 and v_2 defined by the sequence diagram.

Once the CCT and COMDAG are defined, it is possible to define a set of (connected) COMDAG vertices with the smallest number of connections ($\#conn$). To define and design a component and its boundaries select, a set of nodes $\langle A \rangle$ where every node is directly connected to at least one other member in the set. (e.g. a connection is define as $\langle v_n, v_{n+1} \rangle$). Since each node contains a CCT, the newly defined component contains a complete description of the potential interface. A min-cut algorithm can be employed or a designer can manually identify potential component boundaries. In addition, we assume an operational profile has been defined for use cases; that is, each use case is associated with a (relative) frequency. Execution as in [11] or tracing a use case through the COMDAG identifies how often interfaces are activated ($\#act$). A decision on which nodes are part of a defined component are then made based on $(\#conn, \#act)$. The designer can then choose either the interface with the fewest connections, or the interface with the fewest activations. Alternatively, the designer may have set a threshold for $\#act$ and then selected the interface with the fewest $\#conn$ that falls within the threshold. The component $\langle A \rangle$ identified with this approach is a subset of

the vertices in the COMDAG. The activation and connections need not be the only metrics we use. Since the CCT contains attribute and method information, metrics can be collected for class size, method signatures, attribute types, etc. For instance, one of our criteria for a potential component may only select classes that have a maximum of ten methods. The CCTs that define our component will be used in the next section to qualify candidate components.

4. Qualification

This approach determines if an implemented candidate (COTS) component qualifies for the current design and architecture of the system. The analysis is based on comparing the interfaces of the design component as defined in the prior section with a list of implemented candidate components. Interfaces are described in terms of information about the methods, parameters, and attributes as contained in the Constrained Class Tuples (CCT), of Table 1.

Interface analysis determines if an implemented candidate component satisfies the requirements of the system. The set of attributes and methods is described in the form shown in rows 4 and 5 of Table 1, respectively. We assume that a component X may contain multiple attributes and methods, hence many different attribute and method signatures. We will define an interface as comprising multiple attribute and method signatures.

Step 1: The first step to asserting that a candidate component is sufficient for a designed component is to extract the necessary information from the signatures of both the designed and candidate components. Information must be extracted from both the method and attribute sets.

Attribute Extraction, $Attr(X)$, is a function that extracts pertinent attribute information, for component qualification from component X . Component X contains a set of attributes of the form seen in Table 1 row 4. $Attr(X)$ returns the set of all (attribute type, invariant) pairs of component X . We define a metric $\#Attr(X)$ which is equivalent to $|Attr(X)|$.

Method Extraction, $Meth(X)$, extracts pertinent method information, for component qualification from component X . Component X contains a set of methods of the form in Table 1 row 5. $Meth(X)$ returns the set of all (return type, invariant, parameter) triples of component X . We define a metric $\#Meth(X)$ which is equivalent to $|Meth(X)|$.

The method triples $Meth(X)$ and attribute pairs $Attr(X)$ of a component are called its signature.

Step 2: Next we determine if the interface of a candidate component B matches the interface of a designed component A . Each method triple and attribute pair of A is compared to each method triple and attribute pair of B . A *complete match* is found if the signature of the method or at-

| Reference # | Identifier | Definition |
|-------------|-----------------|--|
| 1 | CCT(class name) | $\langle class\ name, \{\{Parent\ CCT\}, \}\{\{Attribute\}\}, \{\{Method\}\}\rangle$ |
| 2 | Attribute | $\langle a\ name, attribute\ type, a\ invariant, \langle CCT \rangle \rangle$ |
| 3 | Method | $\langle m\ name, return\ type, visibility, m\ invariant, \langle Parameters \rangle \rangle$ |
| 4 | {Attributes} | $\{\langle a\ name_1, attribute\ type_1, a\ invariant_1, \langle CCT \rangle_1 \rangle, \langle a\ name_2, attribute\ type_2, a\ invariant_2, \langle CCT \rangle_2 \rangle, \dots, \langle a\ name_m, attribute\ type_m, a\ invariant_m, \langle CCT \rangle_m \rangle\}$ |
| 5 | {Methods} | $\{\langle m\ name_1, return\ type_1, visibility_1, m\ invariant_1, \langle Parameters \rangle_1 \rangle, \langle m\ name_2, return\ type_2, visibility_2, m\ invariant_2, \langle Parameters \rangle_2 \rangle, \dots, \langle m\ name_n, return\ type_n, visibility_n, m\ invariant_n, \langle Parameters \rangle_n \rangle\}$ |

Table 1. A constrained class tuple and its elements.

tribute in A has the same return type, invariant, and parameter tuple or attribute type and invariant of the method or attribute, respectively, in B . A *partial match* is found only if some of the elements of the method or attribute signature in A match the method or attribute signature in B . For a match we need to recursively search through A 's CCT and Parent CCT for methods and attributes whose signatures match the signatures of B (refer to Table 1 row 1).

Step 3: The third step is to determine if the interface of a candidate component B exceeds the interface of a design component A . The interface of B safely exceeds the interface A if it contains enough methods and attributes to match all signatures of methods and attributes in A . Essentially the signature of A , as defined by its attribute pairs $Attr(A)$ and method triples $Meth(A)$, must be a proper subset of the signature of B .

This helps to determine whether or not a given design that requires attributes described by component A can be satisfied by candidate component B . For example, imagine a component A which requires five cards to represent a poker hand. Each of the cards is represented as a String. If candidate component B contains six cards represented as Strings, then component A is a proper subset of component B .

Step 4: We determine a qualification measure. We need to perform operations on sets of attributes and methods of components in order to determine if a candidate component satisfies the requirements. We perform weighted operations to both the attribute and method sets of components. The attribute sets are weighted w_a and the method sets are weighted w_m according to the Analytic Hierarchy Process (AHP) [12]. In the following sections, the fitnesses and coverages computed for both attribute and method sets are also weighted by AHP.

Given two components A and B , with attributes $\{aa_1, aa_2, aa_3, \dots, aa_m\}$ and $\{ba_1, ba_2, ba_3, \dots, ba_n\}$ respectively, the *difference* between component A 's and component B 's attributes is defined in equation 1.

$$\#attrOver(B, A) = \#Attr(B) - \#Attr(A) \quad (1)$$

The difference in equation 1 is the number of attribute elements in $Attr(B)$, but not in $Attr(A)$.

The difference between two components and their attributes is considered the *attribute overhead* of the component. For example the design of a poker game needs a component A which has five cards all represented by Strings, $\{card_1, card_2, card_3, card_4, card_5\}$. Consider a candidate component B which contains six cards all represented by Strings, and a game type represented as an integer number, $\{card_1, card_2, card_3, card_4, card_5, card_6, game_type\}$. Applying $\#Attr(B) - \#Attr(A)$ to the two components results in $\#attrOver = \{card, game_type\}$. The difference in this case represents the *attribute overhead* of the components. Representing the attribute overhead as a percentage is desired, refer to equation 2.

$$attrOver\% = \frac{\#attrOver}{\#Attr(B)} * 100 \quad (2)$$

For this example the percentage of overhead is $2/7 = 29\%$.

Given two components A and B , with methods $\{am_1, am_2, am_3, \dots, am_m\}$ and $\{bm_1, bm_2, bm_3, \dots, bm_n\}$ respectively, the difference between component A 's and component B 's methods is defined in equation 3.

$$\#methOver(B, A) = \#Meth(B) - \#Meth(A) \quad (3)$$

The method difference is the number of method elements in $Meth(B)$, but not in $Meth(A)$. The difference between two components and their methods is considered the *method overhead* of the component. For example the design of a poker game needs a component A which has a deal and shuffle method, $\{getHand(), shuffle()\}$.

If a candidate component B is available which contains methods for `getHand()`, `shuffle()`, and `getBet()`, $\{getHand(), shuffle(), getBet()\}$, then applying $\#Meth(B) - \#Meth(A)$ to the two components results in $\#methOver = \{getBet()\}$. The method overhead of the components as a percentage is defined in equation 4.

$$methOver\% = \frac{\#methOver}{\#Meth(B)} * 100 \quad (4)$$

For this example the percentage of overhead is $1/3 = 33\%$.

The *intersection* of two components A and B , with attributes $\{aa_1, aa_2, aa_3, \dots, aa_m\}$ and $\{ba_1, ba_2, ba_3, \dots, ba_n\}$ respectively, is defined in equation 5.

$$\#attrInt = |Attr(A) \cap Attr(B)| \quad (5)$$

The attribute intersection is the number of attribute elements in both $Attr(A)$ and $Attr(B)$.

The intersection between two components indicates coverage. Thus all attribute elements that are present in $Attr(A)$ and in $Attr(B)$ represent the *attribute coverage*. Attribute coverage can be calculated as in equation 6.

$$attrCov\% = (1 - \frac{\#Attr(A) - \#attrInt}{\#Attr(A)}) * 100 \quad (6)$$

For example, the design of a poker game needs a component A which has five cards all represented by Strings, $\{card_1, card_2, card_3, card_4, card_5\}$. If a candidate component B is available which contains six cards all represented by Strings, and a game type represented as an integer number ($\{card_1, card_2, card_3, card_4, card_5, card_6, game_type\}$), then applying $Attr(A) \cap Attr(B)$, results in $I = \{card_1, card_2, card_3, card_4, card_5\}$. Thus, $|Attr(A)| = 5$ and $\#attrInt = 5$, which indicates that the coverage is $1 - (5 - 5) = 100\%$. However, the attribute coverage must be weighted by w_{ac} . The *attribute fitness* of the component is defined in equation 7.

$$attrFit\% = (1 - \frac{attrOver\%}{100}) * 100 \quad (7)$$

The attribute fitness is $1 - 0.29$ or 71% . The attribute fitness must also be weighted by w_{af} .

The intersection of two components A and B , with methods $\{am_1, am_2, am_3, \dots, am_m\}$ and $\{bm_1, bm_2, bm_3, \dots, bm_n\}$ respectively, is defined in equation 8.

$$\#methInt = |Meth(A) \cap Meth(B)| \quad (8)$$

Where method intersection is the number of method elements in both $Meth(A)$ and $Meth(B)$.

The intersection between two components indicates coverage. Thus all method elements that are present in $Meth(A)$ and in $Meth(B)$ represent the *method coverage*. The method coverage is calculated as in equation 9.

$$methCov\% = (1 - \frac{\#Meth(A) - \#methInt}{\#Meth(A)}) * 100 \quad (9)$$

For example, the design of a poker game needs a component A which has a deal and shuffle method, $\{getHand(), shuffle()\}$. If a component B is available which contains methods for `getHand()`, `shuffle()`, and `getBet()`, $\{getHand(), shuffle(), getBet()\}$, then applying $Meth(B) \cap Meth(A)$ to the two components results in $I = \{getHand(), shuffle()\}$. Thus, $|Meth(A)| = 2$ and $\#methInt = 2$, which indicates that the method coverage is $1 - (2 - 2) = 100\%$. However, the method coverage must be weighted by w_{mc} . The *method fitness* of the component is defined in equation 10.

$$methFit\% = (1 - \frac{methOver\%}{100}) * 100 \quad (10)$$

The method fitness is $1 - 0.33$ or 67% . The method fitness must also be weighted by w_{mf} .

A component B *properly satisfies* a component A if and only if the fitness and coverage is x for both attributes and methods. A component B *satisfies* a component A if and only if the fitness and coverage is at least y for both attributes and methods, where $x > y$ and $x \leq 50\%$. If the fitness and coverage for both attributes and methods is less than y , then a different component should be considered. The sum of attribute fitness and attribute coverage should be as close to 100% as possible to ensure that a given component *attribute qualification* satisfies the requirements of the system. Also, the sum of method fitness and method coverage should be as close to 100% as possible to ensure that a given component *method qualification* satisfies the requirements of the system. Adding the results of the attribute qualification and the method qualification together and dividing by 2, results in the *overall qualification* of the component for the system.

5. Example: Analysis & Qualification

We created a UML designed application to demonstrate the component definition and qualification processes. The application creates a two-dimensional convex polygon filled with either a solid color or Gouraud shading. The inputs to the program are two filenames: an input file and an image file. The input file contains the size, a list of coordinates,

Use Case: Create Polygon
Intent: Create polygon image from input file
Pre Conditions: Input file is correctly specified
Post Conditions: Polygon image file created
Description:
 1. User executes program with 2 command line arguments
 2. System returns polygon image file

Figure 1. Polygon Use Case

and color values. The image file can have five different formats: BMP, GIF, JPG, PNG, or a TIF.

The original class diagram contained 14 classes. Figure 2 shows a simplified version, with many classes hidden behind the image interface. The sequence diagram, Figure 3, has been simplified as well. Both diagrams were designed using UML 2.0 [9]. The COMDAG, Figure 4, was derived from the sequence diagram. The loops in the COMDAG have been retained to reduce the size, but in practice they would be unraveled into a directed acyclic graph.

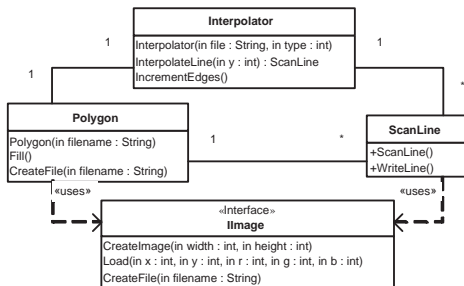


Figure 2. Polygon Class Diagram

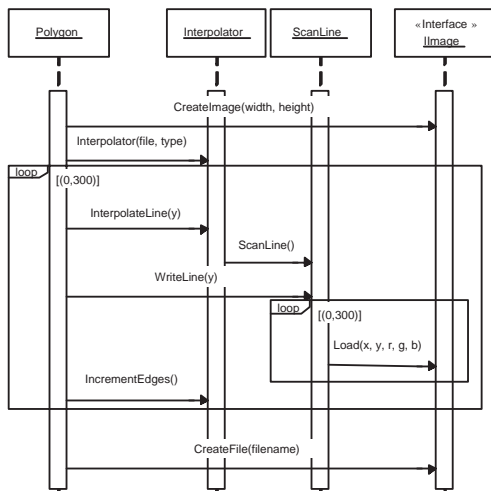
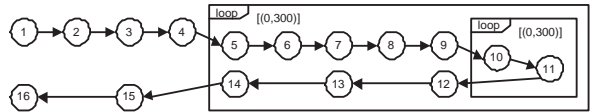


Figure 3. Polygon Sequence Diagram

In our definition process we use an Operational Profile consisting of Use-Cases and their frequency of execution. For this example, we use one of the primary Use-Cases, see Figure 1, to simulate execution. There are three nodes directly connected to the Image Component: 1, 10, and 15. Despite only 3 connections, when we simulate the Use-Case

there are 90,002 activations. Obviously the boundaries to the image interface should also be our component boundaries. Therefore, our component is defined in terms of the CCT's describing image interface.



1. < p, < CreateImage, < 300, int >, < 300, int > >, Polygon >
2. < IImage, < return >, IImage >
3. < p, < Interpolator, < filename, string >, < type, int > >, Polygon >
4. < i, < return >, Interpolator >
5. < p, < InterpolateLine, < y, int > >, Polygon >
6. < i, < ScanLine, null >, Interpolator >
7. < s, < return >, Scanline >
8. < i, < return >, Interpolator >
9. < p, < WriteLine, < y, int > >, Polygon >
10. < s, < Load, < x, int >, < y, int >, < r, int >, < g, int >, < b, int > >, ScanLine >
11. < IImage, < return >, IImage >
12. < s, < return >, ScanLine >
13. < p, < IncrementEdges, null >, Polygon >
14. < i, < return >, Interpolator >
15. < p, < CreateFile, < filename, string, [Type = bmp, gif, jpg, png, tif] > >, Polygon >
16. < IImage, < return >, IImage >

Figure 4. Polygon COMDAG

The purpose of the Image Component is to provide a way for the program to write an image file. We need to be able to specify the image size, the RGB value for each pixel, and create one of five file types. While the tasks the component should perform are quite simple, the component functionality may not conform to our definition. A third party image component may not allow us to create the desired file types, or the interface for loading information may be non-compliant. Other complications arise if the third party component supports extra operations that our program does not use. Although extra functions can be useful, when unused, the excess overhead increases code size without increasing effectiveness.

| | Attr (P) | Attr (IIC) |
|-----------|-------------------------|---|
| Position | int x, int y | int x, int y |
| Color | int r, int g, int b | int r, int g, int b, int a |
| File Type | bmp, gif, jpg, png, tif | bmp, gif, jpg, png, tif, pbm, pgm, ppm, tga |

Table 2. Attributes Required vs. Provided

We will apply Set Analysis to qualify the Imaginary Image Component (IIC), a COTS component. IIC runs on the Java platform, and the interface is provided as a class. The component allows the specification of a generic file, which can be saved as nine different types. IIC allows the image file to be changed one pixel at a time through the specification of a Position and a Color. Although, neither the position nor color class are used in the Polygon specification, we can use CCT's to unravel each structure down to their base types, and perform the comparisons there. Table 2 shows the unraveled attributes. After the image has been created, IIC provides a variety of methods to alter the image

| Meth (P) | Meth (IIC) |
|---|---------------------------------------|
| CreateImage(int width, int height) | SpecifyImage(int w, int h) |
| Load(int x, int y, int r, int g, int b) | LoadPixel(Position p, Color c) |
| CreateFile(String filename) | CreateFile(String name) |
| | <i>Rotate(int degree)</i> |
| | <i>Flip()</i> |
| | <i>Scale(int percentage)</i> |
| | <i>StretchWidth(int percentage)</i> |
| | <i>StretchHeight(int percentage)</i> |
| | <i>Crop(Position tl, Position br)</i> |
| | <i>Invert()</i> |

Table 3. Methods Required vs. Provided

appearance. Table 3 show a complete list of methods in the IIC component.

Everything required by Polygon that is provided by IIC is listed in normal font. By examining Table 2 & 3, it is also apparent that IIC is *sufficient* to cover Polygon's requirements. Because IIC is sufficient, coverage is 100% for both attributes and methods.

$$\begin{aligned} \#AttrInt &= |Attr(P) \cap Attr(IIC)| = 10 \\ AttrCov\% &= (1 - \frac{\#Attr(P) - \#AttrInt}{\#Attr(P)}) \cdot 100 = 100\% \\ \#MethInt &= |Attr(P) \cap Attr(IIC)| = 3 \\ MethCov\% &= (1 - \frac{\#Meth(P) - \#MethInt}{\#Meth(P)}) \cdot 100 = 100\% \end{aligned}$$

Although IIC is sufficient, only 2 methods have a *complete match*. LoadPixel is only a *partial match* because of the extra alpha attribute in the color parameter. All other portions of IIC *safely exceed* Polygon's requirements. The overhead is the portion of IIC that safely exceeds Polygon's requirements. In Table 2 & 3, the overhead is denoted with italics.

$$\begin{aligned} AttrOver\% &= \frac{\#Attr(IIC) - \#Attr(P)}{\#Attr(IIC)} = 33\% \\ AttrFit\% &= (1 - \frac{AttrOver\%}{100}) = 77\% \\ MethOver\% &= \frac{\#Meth(IIC) - \#Meth(P)}{\#Meth(IIC)} = 70\% \\ MethFit\% &= (1 - \frac{MethOver\%}{100}) = 30\% \end{aligned}$$

The component qualification is determined by averaging the coverage and overhead together. In this case the overall qualification is 76.75%, making IIC fairly qualified as a component for the polygon program.

$$Qual = \frac{AttrCov + AttrFit + MethCov + MethFit}{4} = 76.75\%$$

6. Conclusion

This paper showed how to define a design component as part of a UML design. It also defined a method how to use the component interface definition to qualify candidate

components and compute fitness metrics for the degree of fit. An example illustrates how the method works.

Since the method is based on information that can be extracted automatically from a design, it is possible implement a tool that can help in component definition and qualification. This could become a valuable design aid. Automation is our next step.

References

- [1] J. Bieman, L. Ott, "Measuring Functional Cohesion", IEEE Trans. Software Eng., vol. 20, no. 8, pp 644–657, Aug. 1994
- [2] M. Fowler, K. Scott, *UML Distilled Second Edition*, Addison–Wesley, 2000.
- [3] G. Heineman, W. Councill, *Component–Based Software Engineering: Putting the Pieces Together*, Addison–Wesley, Boston, MA, 2001.
- [4] C. Jard, S. Pickin, "COTE - Component Testing Using the Unified Modelling Language", ERCIM News, No. 48, pp 49–50, Jan 2002.
- [5] J. Kontio, "A Case Study in Applying a Systematic Method for COTS Selection", Proceedings of the 18th International Conference on Software Engineering, pp. 201–209, March 25–30, 1996.
- [6] J. Kontio, S-F Chen, K. Limperos, R. Tesoriero, G. Caldiera, and M. Deutsch, "A COTS Selection Method and Experiences of Its Use", 20th Software Engineering Workshop, NASA Software Engineering Laboratory, Greenbelt, MD, 1995.
- [7] C. Larman, *Applying UML and Patterns*, Prentice Hall, 2002.
- [8] N. G. Lester, F.G. Wilkie, and D.W. Bustard, "Applying UML Extensions to Facilitate Software Reuse", UML '98 Beyond the Notation – International Workshop, pp 393–405 1998.
- [9] Object Management Group, "UML 2.0 Draft Specification", <http://www.omg.org/uml>, 2003.
- [10] S. Pickin, C. Jard, T. Heuillard, J. Jezequel, and P. Desfray, "A UML-Integrated Test Description Language for Component Testing", Proceedings PUML Workshop 2001, pp 208–223, 2001.
- [11] O. Pilskalns, A. Andrews, R. France, S. Ghosh, "Rigorous Testing by Merging Structural and Behavioral UML Representations", Proceedings UML 2003, Oct 20–24, pp 234–248, 2003.
- [12] T. Saaty, *The Analytic Hierarchy Process*, New York, NY, McGraw–Hill, 1980.