Performance modeling and analysis of software architectures: An aspect-oriented UML based approach

Kendra Cooper\textsuperscript{a,}\textsuperscript{*}, Lirong Dai\textsuperscript{a}, Yi Deng\textsuperscript{b}

\textsuperscript{a}Department of Computer Science, Mail Station 31, The University of Texas at Dallas, P.O. 830688, Richardson, TX, USA
\textsuperscript{b}School of Computer Science, Florida International University, Miami, 33199, FL, USA

Received 25 March 2004; received in revised form 25 August 2004; accepted 19 October 2004
Available online 13 January 2005

Abstract

Much attention has recently been focused on the problem of effectively developing software systems that meet their non-functional requirements (NFRs). Architectural frameworks have been proposed as a solution to support the design and analysis of NFRs such as performance, security, adaptability, etc. The significant benefits of such work include detecting and removing defects earlier, reducing development time, cost and improving the quality. The Formal Design Analysis Framework (FDAF) is an aspect-oriented approach that supports the automated translation of extended Unified Modeling Language designs for distributed real-time systems into existing formal notations, including Architecture Description Languages Rapide and Armani. The analysis of the formalized design is achieved using existing tool support for the formal methods, which leverages a large body of work in the research community. Currently, FDAF supports the design and analysis of response time and resource utilization performance sub-aspects. This paper presents the algorithms for translating extended UML diagrams into Armani, the proofs of correctness of the algorithms, and an illustration of the FDAF approach by using the Domain Name System. The Armani performance analysis results can provide architects with information indicating whether or not overloaded

\textsuperscript{*}Corresponding author.
E-mail addresses: kcooper@utdallas.edu (K. Cooper), lirongd@utdallas.edu (L. Dai), deng@cs.fiu.edu (Y. Deng).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.10.007
components exist in the design. If such a component exists, then the architect iteratively refines the UML architecture to meet the clients' requirements.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Architecture; Architectural description languages; Non-functional; Unified modeling language

1. Introduction

The design and analysis of non-functional requirements, such as performance, security, accessibility, reusability, etc., at the software architecture design level has received much attention as a means to identify a system’s potential problems (e.g., system’s incompleteness and inconsistency), thus reducing development cost and time while improving quality. The software architecture of a system is the structure of the system which comprises software elements, the externally visible properties of those elements, and the relationships among them [1]. Software architecture raises the level of abstraction and provides designers with the “big picture” of a system by suppressing implementation details.

A system’s requirements include functional requirements and non-functional requirements. Functional requirements are associated with specific tasks or behaviors the system must support. NFRs, also called quality attributes, are constraints on various attributes of the system’s functions or tasks, such as cost, reliability, security, accuracy, performance, modifiability etc. [12]. The important role of NFRs in the development of software systems is that they can serve as selection criteria to help designers with rational decision-making among competing designs, which in turn affects a system’s implementation.

Several approaches have been proposed to develop software architecture to meet the NFRs, including Aspect-Oriented approaches, Unified Modeling Language (UML) based frameworks, and Architectural Description Languages (ADLs). However, many of these approaches focus on handling one NFR at a time and ignore the fact that NFRs interact with each other (e.g., higher security leads to lower performance). Two approaches presented to deal with multiple NFR tradeoff analysis are the NFR Framework [3] and Architecture Tradeoff Analysis Method (ATAM). The NFR Framework supports the requirement analysis phase. The ATAM defines each NFR as an undividable quality attribute and focuses on NFRs tradeoffs that happen during the architecture analysis stage.

Expanding upon the interesting results accomplished by other research groups, the Formal Design Analysis Framework is proposed to support the aspect-oriented design and analysis of multiple NFRs for distributed, real-time systems. FDAF uses and extends part of UML and a set of existing formal methods. The extensions support the automated transformation and subsequent analysis in existing formal methods. Currently, FDAF supports the modeling and analysis of performance aspects. In FDAF, the performance NFR is defined as a set of sub-aspects, including response time, rate throughput, resource utilization, probability of errors, time between errors, durations of event, and time between events [10]. ADLs Rapide [13] and Armani [19] are selected to analyze response time and resource utilization aspects, as these formal notations have well defined performance analysis tools. The algorithms to transform the extended UML into Rapide and Armani
have been implemented in the FDAF tool support. This paper presents the results of using Armani to analyze the resource utilization aspect in a UML based design. Results of using Rapide to simulate the response time aspect are presented in [5]. The Armani performance analysis results can provide architects with information indicating whether or not overloaded components exist in the design. If such a component exists, then the architect iteratively refines their UML architecture to meet the clients’ requirements. The framework is illustrated using the Domain Name System example. The interaction of NFRs is addressed in the next phase of the work using security as the interacting (conflicting) NFR.

This paper, an extension of [6], presents new contributions including the algorithm to translate extended UML sequence diagrams into the formal ADL Armani, the proof of correctness of this algorithm, and a proof of the algorithm presented in [6] that translates extended UML class diagrams into Armani. The remainder of the paper is organized as follows: The FDAF is introduced in Section 2. Section 3 describes the performance aspect modeling and analysis, the translation algorithms, and their proofs. The work is illustrated using the Domain Name System example in Section 4. Section 5 presents a survey of related work. Section 6 presents conclusions and future work.

2. Overview of formal design analysis framework

This section presents an overview of the FDAF. Non-functional requirements tend to interact with each other, for example, increasing the level of security may adversely affect performance. The long term goal of the FDAF is to support aspect-oriented design and analysis of multiple, (possibly) conflicting non-functional requirements using the NFR Framework [3], UML, and a set of existing formal methods.

The FDAF is illustrated in Fig. 1. In the figure, the FDAF is represented with a rectangle surrounded by the stakeholders, inputs and outputs of the framework. Brief descriptions of these are provided below:

Stakeholders. The designers, requirements engineers, etc. who use the framework to develop a system design.

Inputs. Entities used by the stakeholders, including a system design documented in the UML and a requirements specification that includes the system’s functional and non-functional requirements, and existing formal methods may be used in the FDAF.

Outputs. Entities produced by the stakeholders, including a set of aspect-oriented formal design models and the analysis results.

FDAD components. These include an extended UML notation and tool support for FDAF. The tool provides support for defining an extended UML design model, modules to support the translation of an extended UML design model into formal methods, and interfaces with existing formal method tools. A brief overview of FDAF components is presented below:

Extended UML design model. The system architects and designers extend the original UML design with additional, specific details related to the NFRs, creating the extended UML model. An UML extension is needed to support describing NFRs and their automated translation into formal methods. For example, in order to analyze the system’s response time performance aspect, the response time related information needs to be captured in the
UML design model, which is translated into a suitable formal method in a later step of the FDAF.

**NFR framework for design.** FDAF adopts the concepts of the NFR Framework [3]. Here, the NFR Framework is extended to support the design of the system. In this process, NFRs are stated and managed through refining and inter-relating aspects, justifying and documenting decisions, and determining their impact using five kinds of contributions, namely, breaks, hurts, unknown, helps, and makes.

**Use existing formal methods.** The framework assists the designer in selecting formal methods for each non-functional aspect of the system. The tool support for this activity is knowledge based and provides the designer with guidance. Existing formal notations normally are only well suited for describing one or a few types of system properties. By adopting the aspect concept and a set of formal methods, one can select the most suitable notation and analysis techniques for a given aspect.

**Aspect-oriented formal models.** An aspect model focuses on only one type of property, without burden of complexity from other aspects and is potentially much simpler and smaller than a traditional mixed system model. Hence, a set of simpler models, each built for a specific purpose (or aspect) of the system, can be constructed and evolved relatively independently from other aspect models. This allows one to leverage existing understanding of a certain aspect of the system, and dramatically reduce the complexity of modeling and analyzing models.
Formalize extended UML by automated transformation into formal notations. In the FDAF, the extended UML is formalized by translating it into formal notations. An automated translation ensures the consistency between the semi-formal extended UML model and the formal models. Currently, algorithms have been defined and implemented for automating the translation from (part of) extended UML diagrams into Rapide and Armani ADLs.

Analyze the formal model. Once translated, the formal model can be analyzed for specific aspects using its existing tool support. For example, the extended UML diagrams with the response time performance aspect have been translated into Rapide specification. With Rapide's analysis tools available, one can simulate the possible system response time for a particular design, which makes it easier for the architects and designers to evaluate design alternatives. The designer can use analysis results to iteratively modify the semi-formal models and update the formal models. Since NFRs might not be absolutely achieved, they may be “satisficed” as defined in [24], or accomplished in a good enough sense.

3. Performance aspects modeling and analysis

In FDAF, performance is defined as a set of sub-aspects including response time, rate throughput, resource utilization, probability of errors, time between errors, durations of event, and time between events [10]. In the future, additional sub-aspects may be added. Here, we present how two performance sub-aspects, response time and resource utilization, are modelled and analyzed in FDAF. This is followed by a presentation of the translation algorithms and their proofs.

3.1. Modeling and analyzing performance sub-aspects

The conventional UML does not readily support aspect-oriented design. The FDAF proposes a new UML extension to incorporate performance aspect information into a UML model. General abstractions and relationships involved in the performance analysis have been identified and defined as UML stereotypes in the real-time UML [22]. These stereotypes are used in the FDAF and presented in a new graphical notation—a storage-like symbol (grey boxes in Fig. 2).

A set of queuing network theory specific properties are adapted to support performance analysis of software architectures in the Armani analysis tool, namely, they are sLength,
sReplication, sResponseTime, sServiceTime, sUtilization, sVisits. These special properties are also presented in the new symbol. The storage symbol can be associated with UML conventional design elements (i.e., classes, states, stimuli, etc.). The symbol localizes problems as it helps designers to focus on performance aspect elements in an UML aspect-oriented model.

Performance evaluation techniques suggested by [10] are measurement, analytical modeling, and simulation. As the UML does not have such analysis tools, the extended UML aspect models are translated into formal methods Rapide and Armani. Translated UML design can be analyzed by their tool supports. The Rapide ADL is used to simulate the response time aspect and the Armani ADL is used to analyze the resource utilization aspect. This section presents algorithms for the mapping from the extended UML models to Armani. These algorithms have been implemented in the FDAF CASE tool to support automated translation.

Armani focuses on the description of system structure and how the system structure may evolve over time rather than the dynamic run-time behavior of the system. The inputs for the mapping are the UML class diagram and sequence diagram. Each class is translated into an Armani component. Operations of the UML class are translated into ports of the component. As UML attributes and operation parameters are not involved in the performance analysis, they are not translated. Designers should extend UML classes with queuing theory performance properties that are used in the Armani tool, and also provide values for these properties in order to carry out the performance analysis for their design (refer to Fig. 4). There is no need to provide values for all properties since in queuing theory, if two property values are given, the values of other properties can be derived by them. For example, using arrival rate and service time together can calculate utilization and response time. These performance properties are defined as UML stereotypes in the FDAF. Object interactions in the UML sequence diagram are translated into connections between Armani components. The algorithm for translating the UML class diagram and sequence diagram are presented in the following subsections.

3.2. Algorithm for translating an extended UML class diagram into Armani

This section presents the algorithm for translating an extended UML class diagram into an Armani specification. Definitions of UML model elements as well as Armani model elements used in this algorithm are provided. The algorithm is presented in Table 1, followed by its proof.

3.2.1. Algorithm

Definitions used in the algorithm:

1. A UML operation is defined as \( \text{uOperation} = (\text{Name}, \text{Visibility}) \), where: Name is a string, denotes the name of the operation; Visibility is the visibility of the operation. The value of Visibility is ‘public’ or ‘protected’ or ‘private’;

2. A UML association is defined as \( \text{uAssociation} = (\text{End1}, \text{End2}) \), where: End1 and End2 are the names of the two classes that this an association connects. \( (\text{End1}, \text{End2}) = (\text{End2}, \text{End1}) \);
Table 1
The algorithm for translating an extended UML class diagram into Armani

**Input**: A UML class diagram CD, where CD.Classes = \{C_1, C_2, C_3 \ldots C_m\} (\mid CD.Classes \mid = m), CD.Associations = \{A_1, A_2, A_3 \ldots A_n\} (\mid CD.Associations\mid = n);

**Output**: An Armani style specification S, where S.Components = \{SC_1, SC_2, SC_3 \ldots SC_p\} (\mid S.Components\mid = p), S.Connectors = \{SA_1, SA_2, SA_3 \ldots SA_q\} (\mid S.Connectors\mid = q), and satisfies m = p and n = q.

**Translate** (CD) : S

\[
\begin{align*}
S &\leftarrow \emptyset; \\
\text{while } & CD.Classes \neq \emptyset \\
\text{newComponent} & \leftarrow \text{aComponent;} \\
\text{uC} & \in CD.Classes; \\
\text{newComponent.Name} & \leftarrow uC.Name; \\
\text{while } & uC.Operations \neq \emptyset \\
\text{uOpera} & \in uC.Operations; \\
\text{newPort} & \leftarrow uOpera.Name; \\
\text{if } & (uOpera.Visibility == \text{’public’) } \lor (uOpera.Visibility == \text{’protected’}) \\
\text{newPort} & \leftarrow uOpera.Name; \\
\text{Insert newPort into newComponent.Ports;} \\
\text{newComponent.Ports.Length} & \leftarrow newComponent.Ports.Length + 1; \\
\text{newComponent.Ports.Length} & \leftarrow newComponent.Ports.Length + 1; \\
\text{uC.Operations} & \leftarrow uC.Operations - \{uOpera\}; \\
\text{CD.Classes} & \leftarrow CD.Classes - \{uC\}; \\
\text{Insert newComponent into S.Components;} \\
\text{S.Components.Length} & \leftarrow S.Components.Length + 1; \\
\text{while } & CD.Associations \neq \emptyset \\
\text{uAssoc} & \in CD.Associations; \\
\text{newConnector} & \leftarrow \text{aConnector;} \\
\text{Generate a string as the name of the connector;} \\
\text{CD.Associations} & \leftarrow CD.Associations - \{uAssoc\}; \\
\text{Insert newConnector into S.Connectors;} \\
\text{S.Connectors.Length} & \leftarrow S.Connectors.Length + 1; \\
\end{align*}
\]

3. A UML class is defined as uClass = (Name, Operations), where: Name is a string, denotes the name of the class; Operations is a finite set of uOperations;

4. A UML class diagram is defined as uCD = (Classes, Associations) where: Classes is a finite set of uClasses; Associations is a finite set of uAssociation; For each association \in Associations, and association = (class1, class2), class1, class2 \in Classes;

5. An Armani component is defined as aComponent = (Name, Ports), where: Name is a string denoting the name of the component; Ports is a finite list of port for the component. A port is represented by a string which denotes the port’s name;

6. An Armani connector is defined as aConnector = (Name, Caller, Callee), where: Name is a string denoting the name of the connector; Caller and Callee are two roles of the connector;

7. An Armani style specification is defined as a-Style-Specification = (Name, Components, Connectors), where: Name is a string denoting the name of the style specification; Components is a finite list of aComponents; Connectors is a finite list of aConnectors;
3.2.2. Algorithm proofs

Part 1 (Verify the first while loop):
/* Pre: CD is a UML class diagram ∧ |CD.Classes| = X ∧ S.Components.Length = 0 */
while CD.Classes ≠ ∅

... S.Components.Length ← S.Components.Length + 1;
/* Post: X − |CD.Classes| = S.Components.Length */

Proof:
(1) On the initial entry to the loop:

CD.Classes.Length = X ∧ S.Components.Length = 0 (from Pre), and

X − CD.Classes.Length = X − X = 0, S.Components.Length = 0,
thus, P : X − CD.Classes.Length = S.Components.Length is true.

(2) Suppose that P is true before an arbitrary loop iteration. Assume |CD.Classes| = A, S.Components.Length = B, and X − A = B.

The after the iteration: From CD.Classes ← CD.Classes − {uC};

|CD.Classes| = |CD.Classes| − 1 = A − 1
From S.Components.Length ← S.Components.Length + 1;
S.Components.Length = S.Components.Length + 1 = B + 1
Thus, X − (A − 1) = B + 1

(3) On exit from the loop:

P ∧ ¬(CD.Classes ≠ ∅) = CD.Classes ≠ ∅ ∧ P → X = S.Components.Length

→ X − |CD.Classes| = S.Components.Length = post

(4) So long as the loop has not terminated,

P ∧ (CD.Classes ≠ ∅) = X − |CD.Classes| = S.Components.Length ∧
(CD.Classes ≠ ∅) → |CD.Classes| > 0 → t > 0 (t is an integer-valued function)

(5) After each iteration, one element in CD.Classes is deleted from the set. This implies that |CD.Classes| (i.e., t) is decreased by a positive integer amount. Thus the loop will terminate.

Part 2 (Verify the second while loop):
/* Pre: CD is a UML class diagram ∧ |CD.Associations| = X ∧ S.Connectors.Length = 0 */
while CD.Associations ≠ ∅

... S.Connectors.Length ← S.Connectors.Length + 1;
/* Post: X − |CD.Associations| = S.Connectors.Length */

Proof:
(1) On the initial entry to the loop:

CD.Associations.Length = X ∧ S.Connectors.Length = 0 (from Pre), and

X − CD.Associations.Length = X − X = 0, S.Connectors.Length = 0,
thus, P : X − CD.Associations.Length = S.Connectors.Length is true.

(2) Suppose that P is true before an arbitrary loop iteration. Assume |CD.Associations| = A, S.Connectors.Length = B, and X − A = B.
The after the iteration: From \( \text{CD.Associations} \leftarrow \text{CD.Associations} - \{u\text{Assoc}\} \): 

\[ |\text{CD.Associations}| = |\text{CD.Classes}| - 1 = A - 1 \]

From \( \text{S.Connectors.Length} \leftarrow \text{S.Connectors.Length} + 1 \);

\[ \text{S.Connectors.Length} = \text{S.Connectors.Length} + 1 = B + 1 \]

Thus, \( X - (A - 1) = B + 1 \)

(3) On exit from the loop:

\[ \text{P} \land \neg(\text{CD.Associations} \neq \emptyset) = \text{CD.Associations} \neq \emptyset \land \text{P} \rightarrow X = \text{S.Connectors.Length} \rightarrow X - |\text{CD.Associations}| = \text{S.Connectors.Length} = \text{post} \]

(4) So long as the loop has not terminated,

\[ \text{P} \land (\text{CD.Associations} \neq \emptyset) = X - |\text{CD.Associations}| = \text{S.Connectors.Length} \land (\text{CD.Associations} \neq \emptyset) \rightarrow |\text{CD.Associations}| > 0 \rightarrow t > 0 \] (\( t \) is an integer-valued function)

(5) After each iteration, one element in \( \text{CD.Associations} \) is deleted from the set.

This implies that \( |\text{CD.Associations}| \) (i.e., \( t \)) is decreased by a positive integer amount. Thus the loop will terminate.

Therefore, we conclude from Part 1 and Part 2 that the algorithm is correct.

3.3. Algorithm for translating an extended UML sequence diagram into Armani

This section presents the algorithm for translating an extended UML sequence diagram into an Armani specification. Definitions of UML model elements as well as Armani model elements used in this algorithm are provided. The algorithm is presented in Table 2, followed by its proof.

3.3.1. Algorithm

Definitions used in the algorithm (Note: Only stimulus, i.e., asynchronous communication is translated in this algorithm. Future work will consider translating message, i.e., synchronous communication):

1. A UML object is defined as \( u\text{Object} = (\text{Name}, \text{Class}, \text{Performance-Properties}) \), where: \text{Name} is the name of the object and \text{Class} is the name of the class the object belongs to; \text{Performance-Properties} include \( s\text{ServiceTime} \) (float), \( s\text{Visits} \) (float), \( s\text{Replication} \) (int), \( s\text{Utilization} \) (float), \( s\text{Length} \) (float), and \( s\text{ResponseTime} \) (float). All these properties can be associated with values (Note: Performance-Properties are extensions to a UML object for this translation.);

2. A UML Stimulus is defined as \( u\text{Stimulus} = (\text{Sender-Object}, \text{Generation}, \text{Receiver-Object}, \text{Reception}, \text{Performance-Properties}) \), where: \text{Sender-Object} is a \( u\text{Object} \), denotes the instance sending this stimulus; \text{Generation} is a \( u\text{Operation} \) initializing the stimulus and has been defined in \text{Sender-Class}. The Invocation-Style of \text{Generation} is “asynchronous” and its Event-Occurrence is “generation”; \text{Receiver-Object} is a \( u\text{Object} \), denoting the instance receiving this stimulus; \text{Reception} is a \( u\text{Operation} \) receiving the stimulus and has been defined in \text{Receiver-Class}. The Invocation-Style of \text{Reception} is “asynchronous” and its Event-Occurrence is “reception”; \text{Performance-Properties} include \( s\text{DelayTime} \) (float) and \( s\text{Visits} \) (float). They can be associated
Table 2  
The algorithm for translating an extended UML sequence diagram into Armani

### Input:
1. A UML Sequence Diagram SD, SD.Objects = [O₁, O₂, O₃, ..., Oₘ] / Oᵢ is a UML object, SD.Stimuli = [S₁, S₂, S₃, ..., Sₙ] / Sᵢ is a uStimulus;
2. A corresponding UML class diagram CD = (Classes, Associations) for SD, CD.Classes = {C₁, C₂, C₃, ..., Cₜ} / Cᵢ is a uClass}, CD.Associations = {A₁, A₂, A₃, ..., Aₗ} / Aᵢ is a uAssociation;

### Output:
1. I, a list of aInitializations, I = [I₁, I₂, I₃, ..., Iᵢ] / Iᵢ is an aInitialization], and p = m + n;
2. C, a list of aConnections, C = [C₁, C₂, C₃, ..., Cₚ] / Cᵢ is an aConnection, and q = n;

### Translate (SC = ([O₁, O₂, O₃, ..., Oₘ], [S₁, S₂, S₃, ..., Sₙ] ), CD): (I, C)

1. I.Length ← 0;
2. C.Length ← 0;
3. While SD.Objects.Length ≠ 0
   - uObj ← SD.Objects.CurrentElement();
   -uC ← uObj.Class;
   -if uC ∈ CD.Classes
     -aInitial : anInitialization;
     -aInitial.Constituent ← uObj.Name;
     -aInitial.Extended-Constituent ← uObj.Class;
     -Copy uObj.Performance-Properties to aInitial if there is any;
     -Insert aInitial at the end of I;
     -I.Length ← I.Length + 1;
     -Remove uObj from SD.Objects;
   -else
     -Input error;
     -Exit the algorithm;
4. While SD.Stimuli.Length ≠ 0
   -uSti ← SD.Stimuli.CurrentElement();
   -sender-Obj ← uSti.Sender-Object;
   -receiver-Obj ← uSti.Receiver-Object;
   -sender-Class ← sender-Obj.Class;
   -receiver-Class ← receiver-Obj.Class;
   -uAssoc ← (sender-Class, receiver-Class);
   -if uAssoc ∈ CD.Associations
     -aInitial : anInitialization;
     -Name : String;
     -Extended-Connector : String;
     -Extended-Connector ← the name of the Armani connector generated for this UML association in Algorithm A.1;
     -aInitial.Extended-Constituent ← Extended-Connector;
     -Copy uSti.Performance-Properties to aInitial if there is any;
     -Mark uAssoc;
     -aConnec : rConnection;
     -Connec.Sender ← sender-Obj.Name;
     -AConnec.Generation ← uSti.Generation;
     -aConnec.Receiver ← receiver-Obj.Name;
     -aConnec.Reception ← uSti.Reception;
     -Insert aConnec at the end of C;
   -else
     -Input error;
     -Exit the algorithm;
Table 2 (continued)

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.Length ← C.Length + 1;</td>
</tr>
<tr>
<td>Remove uSti from SD.Stimuli;</td>
</tr>
<tr>
<td>SD.Stimuli.Length ← SD.Stimuli.Length − 1;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>Input error;</td>
</tr>
<tr>
<td>Exit the algorithm;</td>
</tr>
</tbody>
</table>

with values (Note: Generation and Reception are two extensions to a UML stimulus. Performance-Properties are extensions to a UML association for this translation);

3. A UML Sequence Diagram is defined as uSD = (Objects, Stimuli) where: Objects is a list of UML objects in the SD; Stimuli is a list of uStimulus, according the order of the time in the sequence diagram.

4. An Armani initialization is defined as aInitialization = (Constituent, Extended-Constituent, Performance-Properties), where: Constituent is a string, denoting the name of an Armani constituent (a component or a connector); Extended-Constituent is a string, denoting the name of an extended Armani constituent; Performance-Properties for components include sServiceTime (float), sVisits (float), sReplication (int), sUtilization (float), sLength (float), and sResponseTime (float). Performance-Properties include sDelayTime (float) and sVisits (float). They can be associated with values.

5. An Armani connection is defined as aConnection = (Sender, Generation, Receiver, Reception), where: Sender is the name of the component that triggers this connection; Generation is the name of the trigger event of this connection; Receiver is the name of the component that receives this connection; Reception is the name of the result event of this connection.

3.3.2. Algorithm proofs

Part 1 (Verify the first while loop):

/* Pre: SD is a UML sequence diagram ∧ SD.Objects.Length = X ∧ I.Length = 0 */

While SD.Objects.Length ≠ 0

...Exit the algorithm;

/*Post: X − SD.Objects.Length = I.Length */

Proof:

(1) On the initial entry to the loop:

SD.Objects.Length = X ∧ I.Length = 0 (from Pre), and

X − SD.Objects.Length = X − X = 0, I.Length = 0,

thus, P : X − SD.Objects.Length = I.Length is true.

(2) Suppose that P is true before an arbitrary loop iteration. Assume SD.Objects.Length = A, I.Length = B, and X − A = B.

The after the iteration, two cases:

Case (i): uC = uObj.Class ∈ CD.Classes:

From I.Length ← I.Length + 1; I.Length = I.Length + 1 = B + 1
From SD.Objects.Length ← SD.Objects.Length − 1;
SD.Objects.Length = SD.Objects.Length - 1 = A - 1; Thus, X - (A - 1) = B + 1

Case (ii); uC = uObj.Class \notin CD.Classes:
    The program terminates; Thus, X - A = B

(3) On exit from the loop:
P \land \neg (SD.Objects.Length \neq 0) = SD.Objects.Length = 0 \land P
\rightarrow X = I.Length \rightarrow X - SD.Objects.Length = I.Length = post

(4) So long as the loop has not terminated.
P \land (SD.Objects.Length \neq 0) = X - SD.Objects.Length
    = I.Length \land (SD.Objects.Length \neq 0) \rightarrow SD.Objects.Length > 0
\rightarrow t > 0 (t is an integer-valued function)

(5) After each iteration, one element in SD.Objects is deleted from the set. This implies that SD.Objects.Length (i.e., t) is decreased by a positive integer amount. Thus the loop will terminate.

Part 2 (Verify the second while loop):
/* Pre: SD is a UML sequence diagram \land SD.Stimuli.Length = X \land C.Length = 0 */
While SD.Stimuli.Length \neq 0
    ... Exit the algorithm;
/* Post: X - SD.Stimuli.Length = C.Length */

Proof:

(1) On the initial entry to the loop:
    SD.Stimuli.Length = X \land C.Length = 0 (from Pre), and
    X - SD.Stimuli.Length = X - X = 0, C.Length = 0,
    thus, P : X - SD.Stimuli.Length = C.Length is true.

(2) Suppose that P is true before an arbitrary loop iteration. Assume
    SD.Stimuli.Length = A, C.Length = B, and X - A = B.
The after the iteration, two cases:
Case (i); uAssoc = (uSti(Sender-Object.Class, uSti.Receiver-Object.Class) \in CD.Associations:
    From C.Length \leftarrow C.Length + 1; C.Length = C.Length + 1 = B + 1
    From SD.Stimuli.Length \leftarrow SD.Stimuli.Length - 1;
    SD.Stimuli.Length = SD.Stimuli.Length - 1 = A - 1; Thus, X - (A - 1) = B + 1

Case (ii); uAssoc = (uSti(Sender-Object.Class, uSti.Receiver-Object.Class) \notin CD.Associations: The program terminates; Thus, X - A = B

(3) On exit from the loop:
P \land \neg (SD.Stimuli.Length \neq 0) = SD.Stimuli.Length = 0 \land P
\rightarrow X = C.Length \rightarrow X - SD.Stimuli.Length = C.Length = post

(4) So long as the loop has not terminated.
P \land (SD.Stimuli.Length \neq 0) = X - SD.Stimuli.Length
    = C.Length \land (SD.Stimuli.Length \neq 0) \rightarrow SD.Stimuli.Length > 0
\rightarrow t > 0 (t is an integer-valued function)
(5) After each iteration, one element in SD.Stimuli is deleted from the set. This implies that SD.Stimuli.Length (i.e., $t$) is decreased by a positive integer amount. Thus the loop will terminate.

Therefore, we conclude from Part 1 and Part 2 that the algorithm is correct.

4. Illustration example

The section presents an example of using the FDAF. The example system is the Domain Name System (DNS) [18], which is a complex system with a rich set of functional and non-functional requirements. It is real-time, distributed, needs to be secure, and (optionally) supports recursive queries. In addition, the DNS is a non-proprietary standard. The DNS provides a way to map a numeric Internet Protocol (IP) address to a character one. IP addresses uniquely identify every computer on the Internet but are hard to remember. The DNS allows networks and hosts to be addressed using common-language names as well as IP addresses and maps host names to various types of addresses through a distributed database.

An application example of the FDAF is illustrated through the DNS system. The base design of the DNS system is captured in the conventional UML. Later, the UML design is extended with performance properties and subsequently translated into Armani ADL specification. The FDAF has developed and implemented algorithms to automatically translate the extended UML design into ADLs Rapide and Armani. This illustration presents the resource utilization aspect analysis results using Armani for the DNS. Examples illustrating FDAF’s support for translating extended UML diagrams into Rapide are presented elsewhere [5].

The UML class diagram for the DNS is presented in Fig. 3, where the Client class represents a DNS client and a DNS server has a Messenger (to decode/encode DNS messages), a Refresher (to refresh database), a Processor (to process clients’ queries), a Monitor (to monitor resource records’ time-to-live), a Generator (to generate queries) and a Database. For simplicity, attributes, operations, and performance properties of classes are ignored. An example of an extended class is presented in Fig. 4 with its Armani specification. The extended class is the class Messenger. It has been translated into a component Messenger in the Armani specification, while the class’s operations have been translated into ports for the Messenger component. This class is extended with a set of
performance properties and each of them is associated with a data value. Fig. 5 presents a UML sequence diagram and its translated Armani specification. In this translation, interactions between objects are translated into Armani connections.

Queueing network modeling has been adapted to support performance analysis of software architectures in [26]. In the approach, software architecture components represent distributed processes and each component has a single queue. Architecture connectors are directional and represent asynchronous message streams. The expected performance of the architecture is derived from performance properties of its components and connectors. Connectors have been considered in the analysis since they can add delays to affect the system’s response time. At the end of the analysis, an additional Boolean property \texttt{sOverloaded} is automatically generated for all components as well as for the whole system indicates whether a certain component is potentially overloaded or not. If the utilization of a component is close to 100\%, this component is very likely to become a bottleneck.
in the system. Fig. 6 shows an example of an overloaded design for the DNS, where a message box shows that the analysis is complete and the design includes overloaded systems. The value of the sOverloaded property for the “Messenger” component is set to “true” which indicates that this component is the potential bottleneck component. The task of this component is decoding received messages, forwarding messages to the correct component, and encoding sending out messages.

There are several ways to deal with a bottleneck component, including replicating the component, decomposing the components to provide estimates at a lower level, speeding the component up, or reducing the demand on the system [26]. It is natural for one to select either one of the first two suggestions to refine the design at current stage to reduce cost without changing the user’s requirements. In this example, the ‘Messenger’ component is replicated (its sReplication property is set to 2) and the new analysis result shows that there is no overloaded component existing (Fig. 7). The original UML class diagram is revised according to the suggestion. Fig. 8 presents the updated design where the multiplicity of the “Message” class is denoted to be 2, which cannot be decided before the analysis.

5. Related work

As the work of the FDAF draws upon aspect-oriented approaches and the extension and translation of UML into the ADLs Rapide and Armani, a brief survey of work in these areas is provided in this section.

5.1. Aspect-oriented approach

High level crosscutting concerns (i.e., NFRs) of a software system are difficult to localize in one single architectural module. Aspect-oriented approaches, based on the separation of the concerns principle, provide an extra decomposition dimension (e.g., crosscutting concern) along which the properties of a software system can be described. The aspect-oriented approach allows the construction and evolution of relatively independent aspect models. Design disciplines Aspect-Oriented Design (AOD) and Aspect-Oriented Software Development (AOSD) have been proposed to provide explicit concepts to modularize the crosscutting concerns and compose these concerns with the system components. The benefits of AOD and AOSD include high cohesion, low coupling and increased modularity in a design. Aspect-oriented approaches focusing on architectural design can be found in [7] and [8].

5.2. Unified modeling language

The semi-formal notation UML [2] is well established in the software engineering community. Its useful capabilities include multiple, interrelated views, and an associated language for expressing constraints on design elements, the Object Constraint Language [27]. It has been considered easier to read and understand than many formal methods. The current UML can be extended with three mechanisms: stereotypes, tagged values, and constraints if it is not semantically sufficient in a certain application. However,
although much of the syntax of UML has been defined, its semantics are mostly described using lengthy paragraphs of often ambiguous informal English, or are missing entirely. As a consequence, UML lacks a formal reasoning ability for its diagrams (e.g., verifying diagrams’ correctness with respect to the system specification) and there is no rigorous analysis tool support for UML.

Two approaches have been used to formalize UML. The first one is the denotational approach [23], which assigns semantics to a language by giving a meaningful representation (e.g., a number, a function, or an already defined formal notation) to its syntactical elements. An example of the approach is to give a denotation to each UML concept in the well-defined formal method Z [4]. The second approach is the translation approach, where (part of) an informal or semi-formal notation is translated into a formal defined notation. An advantage of the approach is that existing valuable verification and validation techniques or tools of the formal target notation can be applied to the source notation. An early example is the formal Requirement Modeling Language (RML) [9], which is originally envisioned as a transformation process from the informal SADT model. One of the UML mappings can be found in [14], where the UML is formally mapped into VHDL. Another example is the Executable and Translatable UML (xtUML) [17] development process, where an application-independent target optimized target specific translator translates application-specific UML models to optimized source code.
5.3. Architecture description language

Architectural description plays an increasingly important role in the process of describing and understanding software systems. A number of Architectural Description Languages have been developed as formal notations to represent and reason about software architectures. ADLs Rapide and Armani are used in the FDAF. Rapide [13] is an event-based concurrent object-oriented language designed for simulation and behavioral analysis of architectures of distributed systems. Rapide is evolved from VHDL, Meta Language, C++ and Task Sequence Language. The semantics of Rapide is partially ordered event sets (POSET). Rapide is presently supported by three kinds of tools for analyzing simulations: constraint checkers, POSET browsers, and animation tools. Armani [19] is a full-fledged
software architecture description language for capturing software architecture design expertise and specifying software architecture designs. The types of architecture design expertise that can be captured with the Armani language include design vocabulary, design rules, and architectural styles. A system’s architecture description specified in Armani is verified by the tool AcmeStudio. One of its built-in tools is software architecture performance analysis based on the queuing network theory. An interchange language ACME [16] has been proposed to support the mapping of architectural specification from one ADL to another (e.g., Wright ADL to Rapide ADL).

5.4. Non-functional requirements approaches

Approaches have been proposed to analyze single NFRs such as performance [25], security [21], and adaptability [20]. The NFR Framework [3] and Architecture Tradeoff Analysis MethodSM [11] are two approaches considering multiple NFR tradeoff analysis. In the NFR Framework, the concept of softgoals is used to represent NFRs and they are related through relationships which represent the influence or interdependency of one softgoal or another. A qualitative analysis method is included for deciding the status of softgoals (i.e., “satisficed”, as defined in [24], denied, conflicting, or undetermined). The framework is applicable in the requirement analysis stage. In ATAM, each NFR has interfaces to other NFRs. These interfaces represent dependencies between NFRs and are defined by parameters that are shared among them. The ATAM helps designers to identify those parameters based on system requirements and no formal methods have been involved in the architecture analysis.

6. Conclusions and future work

Software architecture is an area of software engineering directed at developing large, complex applications in a manner that reduces development costs, increases the quality and facilitates evolution [15]. A central and critical problem software architects face is how to efficiently design and analyze software architecture to meet its non-functional requirements.

The paper presented a UML based approach to model and analyze performance aspects called the FDAF. One contribution of the FDAF is it integrates part of UML and a set of existing formal methods into an aspect-oriented framework at the architecture design level. UML, as a semi-formal notation, is relatively easy for humans to formulate and read. However, its ambiguity makes it difficult for a system’s completeness and consistency checking at the design level. While formal ADLs realize the automated analysis of consistency, completeness, rapid prototyping, and behavior simulation, they are considered more difficult to understand and construct. By integrating part of UML and a set of existing formal methods into an aspect-oriented architectural framework, designers can effectively evaluate and improve their design based on semi-formal models by using existing formal methods tools.

Currently, FDAF provides an extended UML to create designs with performance aspects that cannot be readily analyzed in the real-time version of UML. The paper focuses on the modeling and analysis of resource utilization aspect. The ADL Armani is used to analyze
the aspect based on the extended UML design. The DNS example is used to illustrate the approach. A potential bottleneck component existing in the design has been discovered during the Armani analysis. This result assists architects to identify the problems of their original, UML architecture design in an early stage. The software architecture is then able to be refined iteratively with such analysis results until it is “good enough” to be accepted by the clients, thus reducing development time and cost while enhancing the final system’s completeness and consistency.

There are several interesting directions for the future work of the FDAF. One direction is to investigate the modeling and analysis of additional aspects, such as security, and the interactions among these aspects. The NFR Framework is going to be used to systematically analyze the synergistic and conflicting relationships among the aspects. For example, security and performance represent conflicting NFRs. In general, the more secure a system is, the slower the performance is expected to be, unless alternative solutions such as hardware implementations of encryption/decryption algorithms are considered, which are likely to increase the cost of the system. A second direction is to augment the FDAF aspect repository and improve the corresponding tool support. A third direction is to investigate weaving algorithms to integrate aspect-oriented semi-formal models (UML models that express aspect concern) into a final design.

References


