Correlation of Data in the Unified Modeling Language Interaction Diagram

by

Jonathan Ken Lie

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY June 2000

© Jonathan Ken Lie, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 22, 2000

Certified by

Daniel Jackson
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Correlation of Data in the Unified Modeling Language

Interaction Diagram

by

Jonathan Ken Lie

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A standard was suggested by which modelers can denote dynamic data correlation in interaction diagrams in the Unified Modeling Language. Dynamic data correlation occurs when two or more pieces of data in an interaction either represent the same data or are related. The suggested standard states that correlated data should share the same identifier in the diagram.

Rational PerformanceArchitect was modified to detect, within UML interaction diagrams, any dynamic data that is denoted using the suggested identifier method. After finding correlation, RPA will generate test scripts that cause correlated data to share the same value during test runs of the scripts.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor of Computer Science and Engineering
Acknowledgments

This thesis would not have been possible without the support of several people who I must acknowledge. A special thanks goes to my manager at Rational Software, Mark Richter, whose enlightening ideas and great enthusiasm allowed me to drive forward with this work. Second, I would like to thank Professor Daniel Jackson, whose insightful comments prodded my brain and kept me aware of the theoretical aspects of my company research. Additionally, I would like to acknowledge the following fellow intern and Rational employees for their contributions: Bryan Che for making sure my work made sense, David Seibert for reviewing my documents, Grady Booch for his feedback on my work and for his work on UML, and James Rumbaugh for his comments on data correlation and for his work on UML.

Finally, I would like to thank my family and friends, whose moral support helped me get through my years at MIT.
3.4 Choice for Data Correlation Format: Identifiers

4 Theory of Dynamic Data Correlation in UML – Details

4.1 Correlating Scalars

4.2 Correlating Objects

4.2.1 Arguments for Object Name

4.2.2 Arguments for Association End Rolename

4.2.3 Alternative

4.2.4 Decision on Correlating Objects

4.3 Sample Data

4.3.1 Data Stored in Actors

4.3.2 Data Stored in Use-Case Instances

4.3.3 Data Stored in Interaction Diagrams

4.3.4 Hierarchy of Sample Data Locations

5 Incorporation of Dynamic Data Correlation into Rational PerformanceArchitect

5.1 Issues with the Tools Used

5.2 Process of Incorporating Dynamic Data Correlation

6 Results

6.1 Analysis of the Identifier method

6.2 Analysis of Data Correlation with Rational PerformanceArchitect

7 Conclusion

7.1 Implication of Work

7.2 Future Work

A Generated scripts

A.1 Generated VU code

A.2 Generated Wrapper code
List of Figures

2-1 Simple Sequence Diagram ............................... 17
2-2 Simple Collaboration Diagram ......................... 17
2-3 Example of Data Correlation ........................... 22

3-1 Demonstration of Types of Dynamic Data Correlation .... 26
3-2 Denoting Data Correlation with Parameter Connections ... 28
3-3 Denoting Data Correlation with Message Connectors ....... 30

4-1 Object Correlation with a Common Identifier .............. 35
4-2 Shared Association End Rolenames ...................... 36

5-1 Interaction Diagram with Typical Data Correlations ....... 47
Chapter 1

Introduction

1.1 Importance of Modeling

Software engineers today understand the important roles that solid planning and organization have in producing successful applications. Having well-defined concepts of issues such as the requirements and structure of an application prior to coding a project is key to achieving this success. To communicate these and other software architectural concepts to others working on the project, many people turn to the Unified Modeling Language (UML). Officially adopted by the Object Management Group (OMG) in 1997, UML provides software teams with a common vocabulary with which they can visualize, specify, construct, test, and document their project [4]. Use of a common model truly helps to unify the project team members, along with assisting in organizing an overall plan for the development of the system.

1.2 Importance of Testing

As software projects become more complex, system developers must be on the lookout for a larger number of programming errors, or "bugs." In addition, as these software systems take on an increasingly important role in people's lives, minimizing these bugs becomes a major priority. Thus, project teams are placing a great deal of effort and resources into testing the developed systems to assure a high level of
quality.

1.3 Convergence of Modeling and Testing

Testing oftentimes involves the repeated execution of certain scenarios, which consist of an interaction between one or more objects. One UML construct that is commonly used to describe such behaviors is the interaction diagram. This diagram lays out a possible scenario that a system can undergo, specifying messages and operations that are passed between object instances or classes [9]. Unlike other UML diagrams such as statechart, activity, and use case diagrams, interaction diagrams visually illustrate—in an easy-to-understand format—the manner in which objects interact with each other during execution of the modeled system. The structure of these diagrams is convenient for modeling the behavior of component-based systems, because the diagram details an ordered sequence of messages to and from components. Thus, it is feasible to create—based on some interaction diagrams—executable test scripts that mirror these diagrammed messages. By empowering analysts, developers, and testers with the option to generate test scripts based on models, they are able to begin testing of their designs much earlier in the development cycle than is typically possible.

To make the generated test scripts more realistic, the data involved should be representative of a real interaction. That is, pieces of data in one part of a UML interaction diagram should be correlated with other related pieces of data in another part of the diagram. The UML should have a standard by which modelers can describe data more precisely, from dynamically correlated data to sample data values. Once this standard is established, then tools can then be created to generate test scripts based on the diagram, with the instances of data correlation fully implemented in the scripts. The realization of data correlation in test scripts would lead to a more accurate reflection of the intended behavior of the final system implementation.

The motivation of this thesis is two-fold. The first aim is to make UML a more powerful language by allowing modelers to be more descriptive and more precise with their intentions. Specifically, a reasonable standard is established by which modelers
can denote one specific aspect of system behavior—dynamic data correlation. Along the way, other methods for symbolizing data correlation are considered and compared to the suggested standard. The second aim of this thesis is to enhance the process of generating test scripts that mirror the interactions modeled in a UML model. The proposed enhancement generates test scripts that reflect the data correlation that is present in interaction diagrams, thus yielding more realistic test playbacks. The tool that was developed is part of Rational PerformanceArchitect and is limited to generating VU scripts for testing the performance of implemented COM or distributed COM servers.
Chapter 2

Background

2.1 The Unified Modeling Language

2.1.1 History

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system [9]. Analysts, designers, implementers, and testers of the system share this language, providing a consistent way of communicating ideas with other project members [5].

The UML arose from the collaboration of three of the leading methodologists of the 1990s. Grady Booch, Ivar Jacobson, and James Rumbaugh had developed three separate, popular object-oriented methods, each of which provided a vocabulary for describing different aspects of system specification and development. Eventually, the methodologists began to adopt ideas from each other’s methods, and soon after, all three joined forces at Rational Software Corporation and agreed to create a single, unified language. This unified modeling language would incorporate the best parts of their methods and eliminate confusion between users of the separate methods. After a few years of cooperation between the methodologists and a consortium of several organizations, UML 1.1 was defined and accepted for standardization by the Object Management Group in 1997 [4].
2.1.2 Parts of a UML Model

UML provides a well defined way for expressing the structures and relationships of a system’s objects, the possible ways the system can be used, and even the geographic distribution of the system’s modules. The five interlocking views of a system’s architecture are the use case view, design view, implementation view, process view, and deployment view. The use case view of a system describes the behavior of the system as seen by its users. The design view primarily supports the functional requirements of the system. The process view primarily addresses the performance, scalability, and throughput of the system. The implementation view focuses on how the system is organized into components and files. The deployment view examines the system’s hardware topology. All of these views describe the static and dynamic aspects of the system by using diagrams [4].

The UML diagrams that describe the dynamic aspects of the system are the use case diagram, interaction diagram, activity diagram, and statechart diagram. Each of these diagrams has unique strengths that make it a powerful modeling element, but each also has limitations in the manner in which it describes behavior.

2.1.2.1 Kinds of Behavioral Diagrams

Use case diagrams are important for visualizing, specifying, and documenting the behavior of model elements and are often used to capture the requirements that a system must satisfy. The diagrams contain use cases and actors and indicate the relationships between them. The use cases in a diagram describe what a system (or subsystem, class, or interface) accomplishes but does not specify how it is accomplished [4]. Thus, use cases and the diagrams that reference them can be quite descriptive at a higher level and are not intended to reveal the internal structures of any model entities they describe [9].

Sequence and collaboration diagrams, both of which are called interaction diagrams, articulate interactions among sets of objects as well as the relationships among them. They show sequential orderings of messages between the objects, which
together constitute possible scenarios for the system. Sequence diagrams emphasize the time ordering of messages, and collaboration diagrams emphasize the structural organization of the objects that send and receive messages [4]. These messages and the objects that relay them commonly have directly mappings to operations and classes defined elsewhere in the model. Although interaction diagrams are capable of expressing complex branching behavior, the specification for this behavior may not be visually clear and can be more intuitively expressed in another type of UML diagram called an activity diagram.

Activity diagrams express the flows of control in a system or subsystem from one activity to the next. Each activity represents an ongoing nonatomic execution, which eventually results in some action. An action is composed of a set of computations that changes the state of the system or returns a value. Thus, by specifying transitions between activities, an activity diagram describes the system’s behavior. The diagram supports simple and composite states, sequential branches, and concurrent flows. However, the UML does not prescribe the language of the expressions in the diagram [4]; the syntax for describing an activity can be programming language code, pseudocode, or even natural language [9]. In the common case where natural language is used, humans may understand the intended meaning, but software tools will struggle to parse much meaning from the diagram.

Statechart diagrams illustrate the various states that objects can belong to during the execution of systems, as well as the transitions between those states. A transition from one state to another will take place when a specified event trigger occurs (or when the source state completes its activity) and guard conditions for the transition are satisfied. The transitions and actions that an object can take at a given time, as well as the properties of that object or environment, depend on which state the object is in. Thus, a statechart diagram has much capability in terms of describing the behavior of a single object instance. However, specifying the object’s interactions with other entities can not be done visually, since only one object (via its states) has a visual representation in the diagram. The invisibility of external objects in this respect may make statechart diagrams less intuitive to some novice users.
Of the four kinds of diagrams used for specifying the dynamic aspects of a system, the interaction diagram and the statechart diagram are the best for explaining behavior in a detailed, object-oriented, and visual manner. Previous work has already been done on utilizing the statechart diagram's capabilities for the generation of executable code. For example, Rational Rose RealTime is able to generate functional, executable code from a statechart diagram in an extended version of UML [7]. However, without such a tool, some users may consider statecharts complicated, so another kind of diagram may need to be considered.

No major work has yet been done on tool integration with the UML interaction diagram, despite the diagram's capability to specify behavior rather intuitively and rigorously. Because an interaction diagram shows multiple object instances and describes how they interact, this kind of diagram is ideal for modeling component-based messaging. It would thus seem possible to generate test scripts based on an interaction diagram to simulate things such as the network traffic between the components of an implemented system.

2.1.2.2 Sample Interaction Diagrams

Depending on the circumstance, a modeler may choose between two types of interaction diagrams: sequence diagrams and collaboration diagrams. Sequence diagrams emphasize the time ordering of messages between objects, and collaboration diagrams focus on the objects that interact and the relationships between them. Collaboration diagrams tend to share information with class diagrams, which describe the details of the classes that specify the objects found in interaction diagrams [5]. Both kinds of interaction diagrams contain the same content but have different presentation formats.

The following figures show two interaction diagrams modeling the same scenario—a user withdrawing cash from an ATM. Figure 2-1 shows one of the two types of interaction diagrams: the sequence diagram. Figure 2-2 depicts the other type, the collaboration diagram.
myBankServer: BankServer

myATM: ATM

1. login(AccountNumber, PIN)
   1.1. getAccount(AccountNumber)
   1.2. checkPIN(PIN)

2. withdraw(Amount)
   2.1. withdraw(Amount)

Figure 2-1: A Simple Sequence Diagram

Figure 2-2: A Simple Collaboration Diagram
**Boxes and Stick Figures**  The two diagrams each have two boxes. These boxes are the objects participating in the withdrawal, namely an ATM (myATM) and a Bank Server (myBankServer). MyATM and myBankServer are instances of the classes ATM and BankServer, respectively. As class instances, these two objects have their own identities and properties.

The stick figure in the diagrams is an actor. Actors model users that are outside of the system but interact directly with components in the system. In this case, the actor is a person who uses the ATM.

**Arrows**  The arrows between the objects represent stimuli, which is the UML’s technical term for message instances. The stimuli map to operations that the receiver objects’ classes define. These operations can have argument lists and return values. Stimuli may even have conditional and iteration clauses.

When one object sends a stimulus to another object, it is calling an operation in the receiver object. So, the stimulus login() (the first arrow) represents the ATMUser calling the class ATM’s operation login().

Each stimulus can specify a signature, which is “a string that indicates the name, the arguments, and the return value” of the stimulus [9]. The arguments in the signature apply only to that stimulus, whereas parameters would correspond to the operation that the stimulus maps to.

**Stimuli Ordering**  The message instances in the interaction diagrams are ordered; that is, the stimuli occur within a certain succession. In the sequence diagram, the stimuli are chronologically ordered top-down. There is an implicit time axis that runs vertically along the sequence diagram. Stimuli occurring near the top of the time axis (and diagram) precede stimuli further down the time axis. In the collaboration diagram, stimuli have a Dewey-decimal numbering. The stimuli flow in accordance with this numbering. So, the order of message instances in the diagrams is as follows:

1. MyATM.login()
2. MyBankServer.getAccount()}
3. MyBankServer.checkPIN()
4. MyATM.withdraw()
5. MyBankServer.withdraw()

Note that even though sequence and collaboration diagrams use differing notations to sort stimuli, they still communicate the same ordering.

**Nested Stimuli** In sequence diagrams, arrows point directly to rectangles and proceed from rectangles too. For example, in figure 2-1, `getAccount()` and `checkPIN()`'s arrows stem from a rectangle under `myATM` and end in a rectangle below `myBankServer`. Sequence diagrams use these rectangles to illustrate nested stimuli: `getAccount()` and `checkPIN()` are nested stimuli of `login()`. This means that calling `myATM`'s `login()` operation directly causes `myATM` to call `myBankServer`'s `getAccount()` and `checkPIN()` operations; `login()` consists of `getAccount()` and `checkPIN()`.

Unlike sequence diagrams, collaboration diagrams do not use rectangles to depict nested stimuli. Instead, they use a Dewey-decimal numbering scheme. Figure 2-2 labels `login()` as stimulus number 1. `GetAccount()` and `checkPIN()` are 1.1 and 1.2, respectively. This means that these two message instances are nested stimuli of `login()`. Similarly, stimulus 2.1, `myBankServer`'s `withdraw()`, is a nested message instance of stimulus 2, `myATM`'s `withdraw()`.

**Object Lifetimes** In sequence diagrams, objects have vertical lines running below them. These lines represent the life of the object the objects are valid only for the portion of time for which they have a vertical line below them. In figure 2-1, all the objects have vertical lines below them throughout the entire diagram. Thus, the objects last through the entire scenario.

Collaboration diagrams represent object lifetimes with explicit `<create>` and `<destroy>` stimuli. When an object is created, it receives a `<create>` stimulus. It dies when it receives a `<destroy>` stimulus. Figure 2-2 has no `<create>` or `<destroy>` message instances because the objects last through the entire scenario.
Links

Collaboration diagrams have lines between certain objects. In figure 2-2, for example, there is a line between the \texttt{ATMUser} and \texttt{myATM} and a line between \texttt{myATM} and \texttt{myBankAccount}. These lines are links. Links denote instances of associations or relationships; objects with a link between them are somehow associated or related. Usually, this relationship occurs because one object interacts with the other by passing a stimulus to it.

Sequence diagrams do not explicitly show links. Instead, they imply them: if an object sends a stimulus to another object in a sequence diagram, the sender is implicitly related to the receiver.

2.1.2.3 General Description of UML Interaction Diagrams

Basic Description of Interactions

An interaction diagram depicts, in a visual manner, an interaction between objects in the system. A typical interaction establishes a set of participants, how they are related, and what happens between them. An interaction has a means and duration, as well as a reason for the participants to be involved. Many interactions also take place at a location, at a particular point in time, and for a reason.

For some interactions, the set of participants and the relationships between those objects are more important than the other aspects of the interaction; models usually display these interactions in collaboration diagrams. For other interactions, the aspects of “what happens”, “how the interaction happens”, and “for how long the interaction happens” have greater importance; models usually display these interactions in sequence diagrams.

Specification- and Instance-Level Elements

The interaction diagram elements mentioned thus far have been limited to a specific granularity, namely the instance-level. That is, an object represents an individual entity in the system, and a stimulus represents a particular communication between
two objects. However, interaction diagrams are capable of specifying elements from a more general standpoint. Instead of objects, an interaction diagram can specify classifier roles, which represent a pattern or classification of objects. Likewise, a diagram can specify messages, which describe a pattern of stimuli, and association roles, which describe a pattern of links. Thus, by indicating model entities at this specification-level, an interaction diagram is able to specify a “class” or pattern of specific scenario instances.

The difference between the instance-level and the specification-level can be subtle, but nonetheless it is important [2]. Whereas an object is an instantiation of a class, having attributes assigned with values, a classifier role specifies a restricted view of that class [9]. Objects can conform to the classifier role, thus fulfilling a particular usage of the class. Thus, specification-level diagrams, by using classifier roles, association roles, and messages, describe a template for actual interactions, and instance-level diagrams describe particular objects that relay particular communications.

Every interaction diagram describes an interaction at either the specification-level or the instance-level, but never a mixture of the two. Thus, classifier roles and stimuli never appear on the same diagram, for example. One can differentiate between an object and a classifier role by examining the appearance of the model entity. If an entity’s name (or class name) is underlined, then the entity is an object; otherwise, it is a classifier role. From this, one can determine whether an interaction diagram is defined at the instance-level or the specification-level.

2.1.3 Data Correlation

The data used in the messages of a UML interaction diagram should correspond to data that would be typical of an actual system execution (i.e. a possible scenario). The UML specification suggests a way to elaborate on this data, which appear as object instances and as the arguments and return values of messages. This suggestion implies the use of object references and pseudocode expressions, which can include variable names that are defined within the navigable scope of the sender of the message [9]. Using this suggestion would provide an instinctive way to denote relationships, or
data correlation, between pieces of data in the diagram. The term “data correlation” implies that two or more pieces of data are dependent on each other, so that a change in the value of the first correlated data value results in a change in the other pieces of data. This concept is usually applied to data values that either are identical in value or refer to the same object.

The collaboration diagram in figure 2-3 describes a possible scenario for modifying an employee’s salary, while utilizing the data correlation notation that is mentioned above. In this scenario, an unnamed manager decides to give an employee (whose name is stored in the variable empl_name) a standard raise, so he contacts an object named webApp. After the web application receives the request, it establishes a connection with the database of employees in stimulus 1.1. Once it receives the reference to the database, it tries to ask the same database for an employee in stimulus 1.2. Because the databases involved in stimuli 1.1 and 1.2 are the same, the databases are said to be correlated; therefore, they are both given the same identifier: empDB.

At this point, in the request that webApp sends to the database, the employee name should be the same as the employee name entered by the manager. Thus, the two employee names are correlated, and as such, the arguments share the same variable name: empl_name. In a similar fashion, the employee object returned by stimulus 1.2
and the target object for stimuli 1.3 and 1.4 are correlated and thus use the same identifier. In addition, the value returned by stimulus 1.3 affects the parameter value in stimulus 1.4, so these two data values are correlated as well. Note that the choice to identify objects in the diagram by their rolenames rather than their object names will be analyzed in the thesis.

This identification of data correlation is important for a series of reasons. First, it leads to a more specific model that communicates more information. Second, this information, being stored in a standardized format, can be read and utilized by software tools. For example, by parsing the interaction diagram, a tool could generate a component-based test script that is more realistic of an actual execution of the system than a script without the data correlation would be. Having the capability to test a model this accurately, rather than testing the implemented version of the system, is beneficial for the project team. This is because early testing can detect flaws in the system architecture before time is wasted implementing the faulty design. Considering the importance of data correlation, one might question whether it is possible to use other, more informative, more rigorous, or more efficient methods of indicating data correlation in interaction diagrams.

2.2 Testing

Several testing techniques, such as regression and performance tests, have become popular with software development teams, and companies now offer packages that facilitate this testing process. Performance testing tools, such as Rational PerformanceStudio, can test the load capacity of a variety of servers by simulating thousands of client requests simultaneously [6]. Other tools, like Rational Robot, can record test runs of a GUI interaction or a network communication and play back multiple copies of those runs. A tool recently developed, called Rational PerformanceArchitect (RPA), moves the testing cycle to an earlier stage of the software development stage, namely the point prior to the full development of the client implementation of a two- or three-tiered system. This tool emulates the actions of a client, as modeled in an
interaction diagram of a UML model, by generating analogous test scripts that are then executed against a preexisting server implementation.

2.3 Early Testing

The release of RPA accentuates the importance of performing testing as early as possible, in order to catch errors before much work is wasted on building upon the imperfect design. One common method for determining whether a chosen system architecture or design strategy is feasible is to create a prototype. The prototype is generated quickly and tested lightly, and based on these results, a project team can determine obvious benefits and flaws with the implemented design. To make the prototyping process (and real application development) faster, people can use Rapid Application Development tools, which automate the generation of commonly used code skeletons and code segments. Thus, by having the capability to build temporary or permanent systems quickly, software teams can apply testing techniques to determine flaws early in the development cycle.
Chapter 3

Theory of Dynamic Data

Correlation in UML – Overview

3.1 Forms of Data Correlation in UML

Dynamic data correlation can occur in several forms in an interaction diagram. First, the argument of one stimulus (in this case, something called a dispatching action) can be correlated to one or more arguments of nested stimuli. Second, an object can send correlated data as arguments of two or more stimuli. Third, the argument or return value of a stimulus can be the same as a member variable of the object that is sending the stimulus. Fourth, the return value of one stimulus can be resent (either in the same form or a modified form) as an argument to a future stimulus. Fifth, the return value of a stimulus could be an object that is represented as a box in the interaction diagram. This object may or may not have an association with the object that received the return value. Sixth, any scalar data value found as an argument or a return value can be correlated to a variable used in conditional branches in the diagram.

Each of these cases of dynamic data correlation is demonstrated in sequence in figure 3-1. It should be noted, however, that this sequence diagram would not be coherent if analyzed sequentially from top to bottom.
Figure 3-1: Demonstration of Types of Dynamic Data Correlation
3.2 Desired Features of Data Correlation

One possibly desirable feature is to enforce that if two items of data are said to be correlated, then they will have the same datatype and domain. That is, every correlated piece of data would have a range of values that the data could equal. A difficulty with this feature is the detection of compatible datatypes. For example, an integer can sometimes be cast safely into a floating point number, but because of the declaration of the two items of data, the datatypes are not exactly equivalent, so the correlation may not be permitted. Another difficulty comes in specifying the domains for pieces of data. UML interaction diagrams do not provide fields for message parameters that would allow modelers to specify a domain that an actual runtime argument would have to meet.

Another feature that would be useful for modelers who use computerized modeling tools is the ability to modify a property of one item in a set of correlated data and to have that one change affect all of the other correlated data. In some computerized modeling tools, this behavior is already present for entities such as class names, so having the same behavior for correlated data would make logical sense. However, this would require that parameters be made into higher-ranked objects, with a unique ID number for each parameter. Rational Rose currently does not assign ID numbers to parameters, so this feature would require an important modification in the Rose model specification.

3.3 Rejected Ways to Represent Data Correlation

3.3.1 Argument Connections

One possible way to show that two or more pieces of data in an interaction diagram are correlated is to place a kind of connector between the data representations. Since the pieces of data are oftentimes arguments of messages, the connectors would link individual arguments, as the dotted lines of figure 3-2 do. This connector would be analogous to an association, which connects classes that have relationships with each
other. Just as an association contains information about the relationship between the connected classes, a connector between arguments would also have fields/attributes that would describe aspects of the correlation, such as the domain or sample data values.

Note that these connectors should be connecting arguments, not parameters. Parameters are part of the specification of an operation and is part of the structural definition of a software system. However, dynamic data correlation occurs among executions of the modeled system, which are described behaviorally. Arguments are the runtime versions of parameters, so it is more appropriate to connect arguments in an interaction diagram than parameters.

---

Figure 3-2: Denoting Data Correlation with Parameter Connections
The main problem with this method is its visual conciseness. To denote several data correlations across a single interaction diagram would require several lines connecting the many messages. These additional lines, if numerous, may be messy and could distract from the other lines in the diagram, like object lifelines or links.

3.3.2 Message Connections

Using connectors between arguments may not be the best solution. When multiple arguments exhibit data correlation, the diagram may become disorderly with the number of visible connectors. Also, visually aligning connectors with the specific arguments of a message may be difficult for the modeler or for the reader of the model. One method which would solve these issues would be to connect the messages whose arguments exhibit dynamic data correlation.

Just as UML supports n-ary associations among three or more classes [4], the language could theoretically support n-ary connectors among messages. A connector can have a name, a datatype, and a domain of valid data values that the correlated data can equal. Each end of the n-ary connector is attached to a message and can be referenced by a field analogous to a rolename. This field can identify some aspect of the message participating in the correlation. Another field on the connector end would encode which argument or return value of the message is relevant to the correlation connector. Other information about the correlated piece of data, such as sample data values, can also be stored as part of the connector. An example of how this connector could work can be seen in figure 3-3.

Note that these connectors should be connecting messages, not operations. Operations belong to classes, which constitute a portion of the structural definition of a software system. However, messages specify examples of how objects can interact during runtime. A message can trigger the execution of an object’s operation, specifying a list of arguments. Since dynamic data correlation relates to data during execution of the system, it is more appropriate to connect messages in an interaction
3.3.3 External Database

This method for storing dynamic data correlation information is certainly possible. The main disadvantage is that this information is not readily visible in the model. In fact, the very nature of this method implies that the information is not fundamentally a part of the model. Also, if a plug-in to a software modeling program updates the displayed model with the external correlation data, then keeping the displayed model synchronized with the external correlation database would require effort in terms of implementation and time in terms of real-time processing and synchronization.

3.4 Choice for Data Correlation Format: Identifiers

The general strategy for denoting dynamic data correlation that seems the most justified is a naming scheme. Data values that are correlated should share the same name or identifier, which is specific to the interaction diagram. The presence of any

---

1Connecting stimuli in an instance-level interaction diagram would be just as appropriate as connecting messages in a specification-level interaction diagram.
repeated identifier in the diagram implies that the data values represented by those identifiers are identical.

One benefit of this dynamic data correlation notation is its conciseness. Unlike connectors, identifiers are localized on the interaction diagram and require minimal visual space. Another feature is that this identifier notation is a natural part of the model. Names for pieces of data are stored as part of the interaction diagram.

The UML specification implies a solution along the lines of a naming scheme for dynamic data correlation. With regards to a message signature, section 3.72 of the specification suggests that arguments of a message can use pseudocode expressions that include return values from previous messages [9]. Continuing with this approach, interaction diagrams would treat correlated data as variables with standard names or identifiers. This solution is very intuitive for modelers accustomed to programming, and it requires little or no modification to the current UML specifications.

In comparison to this identifier method of specifying dynamic data correlation, the alternate solutions mentioned above do not appear to be as conventional or “legal” in terms of the UML specification. The proposed solutions involving connectors between correlated data would require the creation of specialized UML elements, an action that would necessitate a strong defense in order to be accepted by general practitioners of UML. The solution of using an external database has several disadvantages, all of which are logistical in nature. Hence, the identifier method will be pursued as the preferred method of denoting dynamic data correlation.
Chapter 4

Theory of Dynamic Data

Correlation in UML – Details

The identifier method of denoting dynamic data correlation applies to several parts of the UML interaction diagram and model. These parts include scalars, objects, and sample data.

4.1 Correlating Scalars

The dynamic scalar data that are involved in an interaction exist either as arguments or as return values of a message. To indicate that two or more pieces of scalar data always have the same value in a particular interaction, the diagram should use the same identifier name to represent each occurrence of the data. If pieces of scalar data do not share the same value but still influence each other, then the representations of those data (usually arguments) can use the common identifier in pseudocode expressions to describe the relationships between the data.

The scope of the identifiers used for denoting dynamic data correlation should be within the interaction diagram, rather than the operations or classes that participate in the diagram. That is, the identifiers should not be restricted to be the same as parameter names or return value names that are defined as part of the operations’ specifications. This gives modelers the flexibility to choose names that are specific to
the interaction diagram and reflect the properties or characteristics of the correlated pieces of data, rather than the properties of the operations.

Additionally, by allowing the identifiers to differ from the corresponding parameter names and return value names, this correlation notation avoids problems with using the predetermined parameter names. For example, if an interaction diagram contains messages that trigger two operations, and both operations happen to have parameters that share the same name, then use of these parameter names would force the appearance of correlation between the two arguments. Using names that are different from the parameter names would solve this problem of denoting correlation falsely.

If the same identifier appears on multiple occasions throughout the interaction diagram, then each instance of the identifier matches a piece of data that is correlated with the rest of the instances. One exception of this occurs when attribute names of different objects happen to be identical. However, if these attribute names are always referenced along with the parent object's name, then the resulting fully-expanded names will not match, thus signifying no correlation.

One habit to be aware of is that of reusing temporary variable names. Programmers may sometimes store different transient values into the same variable during a period of time; this style of naming is discouraged in regards to denoting dynamic data correlation. Although reassignment of new return values to the same variable is legal in most programming languages, the proposed data correlation method would treat the data of each reassignment as correlated pieces of data. Thus, modelers should avoid using the same identifier to store multiple return values. Note that this also holds true for return values that are objects.

4.2 Correlating Objects

Object correlation is a condition in which two or more occurrences of objects represent the same entity. The presence of object correlation in modern software systems is quite common. That is, in a typical object-oriented system, an operation returns an object
reference to a central entity, which in turn delegates a subsequent operation to that returned object reference. The object correlation that is present in this situation links the return value of the first operation with the target object of the latter operation; this correlation implies that the two objects are the same.

Object correlation is also visible in UML interaction diagrams that model such systems. In an interaction diagram, an object can appear in three major forms: an object reference passed into a message (as an argument), an object reference returned from a message, and an object participating in a collaboration. To indicate the linkage between two or more objects, a modeler should use a single identifier to reference the multiple occurrences of the single object. The identifier can be used in the return-value portion of the message signature (or in a return-value message), in the object’s “name” field, or in any other location where the diagram may denote dynamic data correlation.

In figure 4-1, the identifier used as the return value of the first stimulus matches the name of the instance belonging to the class CDocument. The common identifier used to denote the object correlation is “theDocument”.

Selecting the appropriate identifier for denoting correlation of an object in the diagram is a subtle point. In some cases, using the object name seems to be the most relevant choice; other times, an association end rolename appears to be more proper. UML practitioners can make valid arguments for using one or more of these language
elements for housing the correlation identifier, and below are listings of key points for the two main contentions.

4.2.1 Arguments for Object Name

First, an object name, compared to an association end rolename, is more likely to provide unique identification of a navigable object. With object names, a modeler can ensure that each object represented in the interaction diagram has a distinct identifier. However, this is not always the case for association end roles. For example, if an object A in an interaction diagram is linked with two objects of the same classifier by the same kind of association, then the rolenames that refer to the two objects are identical. Thus, object A can not distinguish between the two objects if association end role names are used.

In figure 4-2, the objects named Jane and John belong to the same classifier (Person) and have the same kind of association with the object named ProfSmith. If ProfSmith refers to the other objects by object name, no confusion arises; however, if association roles are used (e.g. for denoting dynamic data correlation), then the objects become indistinguishable. Thus, in this case, object names are better suited for providing unambiguous identifiers for denoting object correlation.

Conversely, a less common, yet still possible, case involves two objects that share two or more links. In this case, the one target object may be referenced through
any of the two or more links and association end rolenames. This happens when the
target object plays different roles from the source object's perspective. (For example,
a linked list object can be associated with a single node through two links, where
one association end rolename for the node is “head” and the other is “tail.”) If the
results of a message sent to the target object are dependent on the role that acts upon
the message, then it is essential to identify the role that the message corresponds to.
Thus, using the object name seems to lead to ambiguity. Furthermore, if the data
correlation pertains to the target object’s role, then it appears that the rolename
should be used to denote correlation instead of the object name.

However, in programming languages like Java and C++, association end roles have
no direct correspondence to language constructs. Instead, the intentions of an associ-
ation between a source object and a target object are encoded in the implementation
and state of the target object. For example, consider a child who is both a student
and an offspring of an adult. In UML, a message instance of discipline() to the stu-
dent rolename implies different behavior than the same message instance sent to the
offspring rolename. In languages like Java and C++, however, this action is generally
coded as theAdult.student.dicipline() or theAdult.offspring.dicipline(),
but student and offspring point to the same object. Assuming no multiple inheri-
tance of implementation in C++, the difference in behavior is not determined from
the rolename but rather is determined inside the one implementation of the function
dicipline, based on the current state of the student/offspring. Therefore, in these
languages, knowledge of the association end role is important only when implement-
ing the target object’s functions, not when calling the object’s functions. Once the
functions are implemented, programs access them through the object instance, rather
than through an interface; likewise, interaction diagrams should refer to objects by
the object name, not by an association end rolename.

Second, object names are specific to the collaboration they are used in; similarly,
the dynamic data correlation that this paper refers to applies only to one specific
interaction or collaboration at a time. However, association end rolenames are defined
at the broader specification level and can apply to all diagrams in a model. Thus,
from the specification-/instance-level perspective, object names are more appropriate than association end rolenames for representing dynamic data correlation.

Third, an association end role is generally assigned a name that encompasses the descriptions of all the possible instances of the role. In order to accomplish this, the rolename must be somewhat general. In contrast, object names are assigned in specific collaborations, and as such, they have the power to be more descriptive than a corresponding association end rolename. In this way, use of the object name provides the flexibility to describe data more precisely for a given interaction. Since correlation identifiers would benefit from this descriptive power, the object name would serve better as correlation identifier than the association end rolename would.

4.2.2 Arguments for Association End Rolename

First, figure 3-50 in the UML specification shows an example where the rolename of an association end is used as an argument to a stimulus [9]. This is a form of object correlation.

Second, the rolename at the target end of an association “provides a name for traversing from a source instance across the association to the target instance or set of target instances. It represents a pseudo-attribute of the source classifier” [9]. In this sense, the association end rolename provides the source instance with a hook by which to identify associated objects. Thus, when a link is formed between the source instance and an object that a stimulus returns, then the source instance can refer to the returned object like a pseudo-attribute.

This use of the rolename is similar to how the dynamic data correlation identifier is used for scalars. Whereas the source instance accesses a correlated object through an association end rolename, the same source instance would access correlated scalar data through a correlation identifier. Whereas the rolename is a treated as a pseudo-attribute of the source object, the correlation identifier is treated as a locally visible variable. Because pseudo-attributes are forms of locally visible variables (within the scope of the source instance), a natural extension would be to use the target association end rolename as the dynamic data correlation identifier for objects.
4.2.3 Alternative

One could argue for the use of classifier rolenames rather than object names. The arguments for these two sides would be very similar, with main differences being in the proper semantic uses of objects and classifier roles as prescribed by the UML.

4.2.4 Decision on Correlating Objects

The UML makes a strong distinction between the specification level and the instance level. The dichotomy between classes and objects, associations and links, etc. is pervasive through the definition of the language. As such, it seems prudent for the data correlation notation to conform to this standard. The UML indicates that object names exist in the instance level, whereas association end roles are not specified at that level. Because interaction diagrams tend to deal with dynamic data correlation at the instance level, the use of object names seems to be more appropriate for dealing with the correlation.

To the author, the arguments for the association end rolename, though quite valid, seem less rooted in the overall spirit of the UML specifications than the arguments for the object name. From this analysis, the author concludes that object names are preferable to association end rolenames for use as object correlation identifiers.

However, in certain situations, association end rolenames would seem to be the appropriate choice, namely when an object's attributes or pseudo-attributes are being modified or correlated. Thus, a modeler should try to use object names when possible, except where association end rolenames precisely specify the intended object by way of an attribute.

4.3 Sample Data

A UML interaction diagram can show a possible scenario of objects passing stimuli back and forth. In an instance of the scenario, the objects pass stimuli that contain specific data values as arguments. In other words, an interaction diagram is permitted
to display real data values as the arguments of stimuli.

However, when denoting dynamic data correlation, an interaction diagram uses identifiers as the arguments of stimuli. This causes a conflict in regards to the capability to display both identifiers and sample data values in the diagram. Several possibilities exist for displaying these data values in a UML model, many of which were proposed by Mark Richter of Rational Software.

4.3.1 Data Stored in Actors

It is conceivable for actors to have properties. From a practical standpoint, actors may have identification numbers, names, or preferred withdrawal amounts. From a language standpoint, actors (as do all UML elements) support tagged values, and an actor’s properties can be defined in this manner.

Actors act on use cases, which may have interaction diagrams attached to them. To relate an actor’s properties to such an interaction diagram, the name of the tagged value would match an identifier found as an argument in the diagram. One benefit of keeping the identifier in the argument position is to permit notation of data correlation. Another benefit is the ability to vary the sample data values (by applying a number of related actors with different tag values) without modifying the interaction diagram.

This method should not be used, however, because the concept is not self-consistent. Sometimes, properties of a specific actor will vary; for example, the amount of money an actor (person) may want to withdraw can change from transaction to transaction. This variable data is typically what is inputted into interaction diagrams, and forcing this variable sample data onto a static actor is not appropriate (since the solution would be to create multiple versions of that one actor in the model, each with different tag values).
4.3.2 Data Stored in Use-Case Instances

The purpose of use cases is to specify possible uses of a system, subsystem, etc. in a general manner. Giving real data values in a use case would make the use case too specific, so a better alternative to this would be to consider use-case instances.

Use-case instances can be thought of as realizations of use cases. The realizations have actual data instances rather than specifications for data values. Therefore, it would be appropriate to have sample data stored as tagged values of a use-case instance. As with the actors, the tag names would correspond to identifiers found in an associated interaction diagram, and the tag values would contain the real (sample) data values.

4.3.3 Data Stored in Interaction Diagrams

Arguments in an interaction diagram can be real data values, identifiers, or pseudocode. However, the UML specifications do not mention the possibility of denoting any two of these together as the argument. Thus, it would not be possible for an interaction diagram to use an argument to show both a sample data value and correlation with other data.

An extension to UML would allow the diagram to convey such information. By allowing arguments to take the format

\[
\text{identifier} : \text{type} = \text{value},
\]

the diagram would be able to show both dynamic data correlation and sample data values.

This format, though not an official format used in interaction diagrams, is analogous to the format used for specifying the name, data type, and default initial value for class attributes [4]. Thus, it is natural to extend this same format to arguments of messages in interaction diagrams.
4.3.4 Hierarchy of Sample Data Locations

Rather than selecting a single place to specify sample data values, Mark Richter suggests finding sample data based on a hierarchy of locations [8]. Any sample data stored in the use-case instance that owns the interaction diagram will have precedence. If no data is found there, then the next location for sample data would be in the message signatures in the interaction diagram. In the case that data is absent in both of these locations, then one can use either the default value for the parameter (if applicable) or a default value for the datatype of the data in question.
Chapter 5
Incorporation of Dynamic Data Correlation into Rational Performance Architect

After developing the theory of the proposed identifier method of denoting dynamic data correlation, it was necessary to determine its practicality. That is, if there existed a software program that could utilize the notion of dynamic data correlation, then that program could be modified to incorporate that notion in order to confirm the theory's applicability.

Of the existing software programs that allow users to draw and manipulate UML models, Rational Rose was the most readily available program to the author. In addition, Rational Rose happens to be the leading tool for UML modelers and has several features that proved to be extremely helpful. Rational Rose supports the development of add-ins that can be run from the menu bar; these add-ins can communicate with Rose using an application programming interface known as the Rose Extensibility Interface (REI). Using REI, add-ins, as well as external programs using Automation\(^1\), can read Rose models by traversing the hierarchy of the model (between classes, operations, diagrams, etc.). Because of these points, Rose was chosen

\(^1\)a special way of executing commands in other applications on Windows platforms
as the target modeling application for the tests on implementing the processing of dynamic data correlation.

A summer before work began on dynamic data correlation, a product called Rational Performance Architect (RPA) was developed. RPA is an add-in to Rational Rose that allows analysts to design client-side interactions with a server and to generate analogous test scripts that can be run against the server. Dynamic data correlation is very relevant for RPA because the generated test scripts could be made to convey more realistic data during playback. Therefore, efforts to add modularized modifications to the RPA source code seemed to be a good option.

RPA is implemented in RoseScript, a dialect of BasicScript, which is similar to Visual Basic. RoseScript has built-in REI commands, which facilitate the construction of utilities for manipulating Rose models. When the RoseScript code for RPA is run, the resulting generated scripts are written in VU, a light version of C with special commands. VU scripts can be executed in Rational LoadTest, a software tool that can test the load capacity of a variety of servers. RPA was written to process UML models of COM-based client-server systems and the interactions between the client and server, so the generated VU scripts try to emulate the COM interactions. However, VU does not have support for calling COM methods, so the workaround that RPA uses is to resort to generating a dynamically-linked library as an intermediate entity. This DLL, whose functions can be called from a VU script, can act as stubs that perform the actual COM interactions. Thus, RPA generates two scripts: one for the LoadTest application (written in VU) and another that is compiled into a DLL (written in C++, with COM support).

5.1 Issues with the Tools Used

Although the number of supported features in the Rational Rose program is ample for most modeling needs, a few minor details made incorporation of dynamic data correlation inconvenient. First, the program does not support the UML signature na-

\[\text{Component Object Model}\]
Message arguments default either to the corresponding parameter's datatype, name, or datatype and name. Modification of the argument is saved as part of the "message name," rather than the message signature. As a result, accessing any one argument had to be accomplished by parsing the free-form text of the message name.

Second, the application does not support the notation of conditional branches in interaction diagrams. Thus, this thesis did not explore data correlation with respect to conditional statements.

Third, the Rose Extensibility Interface is not yet able to access everything that is presented in the Rose user interface. For example, a script using REI can not navigate into a nested diagram located in an interaction diagram. This makes the enforcement of spanning dynamic data correlation across multiple diagrams difficult. Additionally, the REI can not determine which messages are nested. Thus, if the execution of one operation triggers the execution of a second operation, then the REI will treat the two operations as being independent from each other. The inability to distinguish nested messages may cause test script generators to replicate the execution of a nested operation.

Last, RoseScript is a scripting language that is intended for lightweight programmable tasks. RPA is already implemented in RoseScript, so the implementation of dynamic data correlation detection is also in RoseScript. However, the added correlation feature would benefit from a stronger, scalable language. The RPA development team has discovered some errors which may be a result of a 16-bit language. Using the Automation interface of REI would be advisable, because this would allow use of another programming language as the language of the source code.

---

3 First, strings must not exceed approximately 65536 characters in length in order for RPA to run properly. Second, in the file containing the source code of RPA, functions that exist more than sixty- to seventy-thousand bytes apart can not call or return to each other.
5.2 Process of Incorporating Dynamic Data Correlation

The ability to modify the source code of the Rational PerformanceArchitect plug-in eased the implementation process greatly. With the fundamental structure of converting interaction diagrams into test scripts in place, the task of implementing a tool that finds and utilizes dynamic data correlation became easier. More attention could be placed on the details of the correlation rather than the test script generation.

Without RPA as a test script generator, a tool that finds and utilizes data correlation would be useless. In fact, the translation of correlation from the interaction diagram to generated scripts is merely an enhancement for the scripts. These scripts are fundamentally designed to reproduce the interactions between clients and COM servers. Thus, when RPA generates the function calls in the generated script, it must find the messages in the interaction diagram that are sent to COM objects.

Upon identifying the relevant COM messages in an interaction diagram, the script generator should search for identifiers that play a role in denoting data correlation. These correlation identifiers should be collected, analyzed, and used to make the generated scripts reflect the same data correlation as the diagram. In many respects, this action parallels the functionality of a compiler [1]. This tool first parses a language, which is the UML interaction diagram in this case. Next, it tracks the identifiers used in the diagram, storing the information found into a low-level representation (possibly similar to a symbol table). Finally, it generates another form of code, which happens to be a VU script, rather than assembly code. In this way, the enhanced version of RPA will convert one representation of dynamic system behavior into another.

One important change that must occur to the script generator is that it must make at least two passes over the messages or the corresponding generated code, in order for the correlation data to propagate through. The reason for this is because of the method chosen for denoting data correlation. To determine if an item of data is correlated to any other piece of data in the diagram, RPA must scan through all the relevant identifiers to find a match. At the completion of this first scan, the script
generator will have collected the correlation information, so a second pass is needed to incorporate this information into the generated scripts.

One method for implementing this double pass is to have the initial pass over the messages collect the correlation information, storing it in a sort of external database. During the second pass, the messages will be translated into VU code, with the correlation information incorporated. A second method for implementing the two passes is to generate a temporary copy of the scripts (as with the original RPA) while simultaneously finding correlation. The second pass would then parse the generated scripts and edit these to incorporate dynamic data correlation. Because line-by-line editing tools (such as awk in UNIX) appear not to be widely used in the Windows platform, the first method was chosen.

```
1. vu_CoInitialize()

2. PType_Int(iNum : int = 42, &retval : int*)

3. getLog(&theLog : ILogger**)

4. write(retval, rc : int*)

5. vu_CoUninitialize()

Figure 5-1: Interaction Diagram with Typical Data Correlations
```

To test the implementation of the dynamic data correlation module in RPA, scripts were generated from the interaction diagram shown in figure 5-1. The first stimulus performs some necessary COM initializations. The next stimulus tests a scalar data
correlation, as well as a sample data value in the diagram. Stimulus #3 attempts to denote object correlation, and the following stimulus exhibits more data correlation. The final stimulus performs some necessary COM uninitializations. The VU script and COM wrapper that RPA generates for this diagram are included in appendix A. When the scripts are executed, the dynamic data correlation encoded in the interaction diagram occur. This is evident in the manner in which the VU script uses variables to store the values of correlated pieces of data, such as retval and theLog.
Chapter 6

Results

6.1 Analysis of the Identifier method

The theory behind the proposed method of denoting dynamic data correlation appears to be quite viable. In comparison to other methods considered, the one outlined in this thesis seems to be the best choice from a general viewpoint. However, some deficiencies in the proposal stand out as well.

Benefits

One of the strong points of using common identifiers to designate correlated data is its usability by modelers. The method of having one identifier refer to one item of data is intuitive for many programmers, who use variable names to identify data. The method is also preferable to the other proposed methods because of its strong visibility and clarity within the model. Use of text avoids the issue of numerous lines running across diagrams. In addition, unlike the external database method, the identifiers used for denoting dynamic data correlation are stored as part of the UML model.

Another advantage of this identifier method is its strong compatibility with the existing UML specification. UML does not prescribe any format for message arguments in interaction diagrams, but instead encourages modelers to use expressions
that seem appropriate. Using identifiers for dynamic data correlation and the format identifier : type = value to display sample data values is thus legal in UML.

**Problems**

A problem with the identifier method is that low-class UML objects are involved in the notation of dynamic data correlation. Whereas UML objects such as classes, associations, and sometimes operations generally have a higher level of importance, arguments and the free-form text used to denote them are less important. As a result, some software tools and packages may not bother to include much support for argument strings. For example, Rose does not assign unique IDs to parameters, and REI is unable to retrieve one argument from the argument list of a message. However, classes have unique ID numbers assigned to them in the underlying storage format of the Rose model, and the REI provides several means by which one can access the classes of a model.

Because the identifiers used for the arguments are not navigable using REI, modelers can not modify a collection of correlated pieces of data simultaneously. If another method were chosen to denote data correlation, such as the use of connectors between pieces of data, then the model could store and modify properties of the correlated data entity. However, for the identifier method, modelers will have to modify each argument in order to propagate a change.

### 6.2 Analysis of Data Correlation with Rational Performance Architect

Use of the identifier method for notating dynamic data correlation should be easy and straightforward for modelers. A modeler needs only to modify the message signature in an interaction diagram to convey the concept of correlation. Most model-editing tools like Rational Rose permit this operation.

However, transferring the data correlation and the sample data values from the
interaction diagram to generated scripts is not perfect. There is the question of whether the generated scripts, when played back in Rational LoadTest, will emulate the timings of a real client properly. Examination of the generated scripts would imply that the scripts should work functionally, but the performance of the scripts has not been determined. In particular, the generated wrapper code in appendix A shows some runtime checks, `AddRef()` and `Release()`, which determine if a target object is truly a COM object. However, execution of these operations would increase traffic to the COM server by up to three times. A better generated script would not run these checks but would instead catch any error that was produced by assuming that the target object has a COM interface.

Another deficiency exists in this work’s specific implementation of the RPA module that deals with dynamic data correlation and sample values. This deficiency, with respect to the UML, is that the module enforces the use of C syntax for specifying strings as sample data values. The reason for this requirement is to allow the script generator to create scripts that would use the strings intended by the modeler. UML does not specify how backslashes, tabs, or newlines should be encoded in text, so the module presumes that strings are preformatted. The module could have been modified to establish a language-unspecific format for denoting strings and then to generate the appropriate VU strings from this format. However, this seemed to require more effort than was necessary.
Chapter 7

Conclusion

7.1 Implication of Work

This thesis began by examining the possibility of denoting dynamic data correlation in the Unified Modeling Language interaction diagram. Several possibilities were considered, but the method that was the most logical was the use of identifiers, or names, to specify that pieces of data are correlated. Using identifiers to denote data correlation fits within the specifications of UML and does not require any additional UML constructs, such as connectors between arguments or messages in interaction diagrams. With dynamic data correlation in place, the interaction diagram is capable of conveying more information about the data that gets passed between objects in an interaction.

Also considered was the possibility of storing examples of data values. Because these data values can be placed in a number of locations within a model, it was decided that a hierarchy of locations would be established. Any data found in a use-case instance overrides data found in the argument of an interaction diagram, which in turn overrides default initial values for parameters. Again, as with dynamic data correlation, the ability to specify sample data values in an interaction diagram and in the model helps modelers to convey their intentions more accurately than without data values.

Finally, the notions of data correlation and sample data values were incorporated
into an existing software tool called Rational PerformanceArchitect. The enhancements to RPA allowed the Rational Rose add-in to generate test scripts that better reflected real-world transactions, with more realistic data.

7.2 Future Work

One possible area towards which one could extend this research is the notion of dynamic data correlation between multiple nested or concurrent interaction diagrams. For nested diagrams, use of a common identifier seems reasonable because of the linear, sequential nature of the execution of the nested diagram within another diagram. However, correlation between concurrently running interactions may pose some challenges, because each diagram may run separately. Understanding the scope of the identifiers used in a given diagram may become an issue.

Another topic to consider is the definition and datapooling of domains. Sometimes, one will want to restrict the set of values that a particular piece of data can take (for example, non-negative numbers), yet creating a new type or class would be excessive. For operations, preconditions can handle this task, but for pieces of data that are dynamically correlated across several messages in an interaction diagram, specifying the domain is not as straightforward.

A third topic, as suggested by Grady Booch [3], would be the analysis of a mathematical mapping between interaction diagrams and statecharts. Both describe the execution of functions in a sequential manner, but this thesis focused solely on interaction diagrams because the consensus at Rational is that interaction diagrams are easier than statecharts for new modelers to learn, understand, and use. Perhaps as UML becomes more widespread, and as more modelers become comfortable with statecharts, then Rational PerformanceArchitect and dynamic data correlation can find their way to that part of the UML.
Appendix A

Generated scripts

A.1 Generated VU code

```c
#include <VU.h>

externC string strResult; /*for string results*/
externC int intResult; /*for non-string results*/
externC string pszLogPass; /*log message for success*/
externC string pszLogFail; /*log message for fail*/
externC int func vu_ColumnInitialize() {} /*initialize COM*/
externC int func vu_ColumnUninitialize() {} /*uninitialize COM*/
externC int func IDatatypes() {}
externC int func IDatatypes_PType_Int(Param_int, plObj)
string Param_int;
{}
externC int func IDatatypes_getLog(plObj)
{}
externC int func ILogger_write(bsVal, plObj)
string bsVal;
{}
/* Main */
{
    string retval;
    int pLong[1];
    int NumberCruncher;
    int theLog;
    /* ******************************************* */
    /* The following code is common initialization code for all scripts */
    /* ******************************************* */
    push Timeout_scale = 200; /* Set timeouts to 200% of maximum response Time */
    push [Think_avg = 0, Think_dist = "NEGEXP", Think_def = "LS"];
    push Timeout_val = 120000; /* Set minimum Timeout_val to 2 minutes */
    Pfunctions = datapool_open("Pfunctions");
    datapool_fetch(Pfunctions);
```
/* Operations mapped in the sequence diagram */

vu_CoInitialize(); /* Initializes COM - maps to vu_CoInitialize() in sequence diagram */

emulate ["IDatatypes_PType_Int2" ] IDatatypes_PType_Int(
    datapool_value(Pfunctions, "iNum")),
    pszLogPass, pszLogFail;
    retval = itoa(intResult);
    pLong = addr_plDatatypes();
    NumberCruncher = pLong[0];
    emulate ["IDatatypes_getLog3" ] IDatatypes_getLog(
        NumberCruncher),
        pszLogPass, pszLogFail;
    theLog = intResult;
    emulate ["ILogger_write4" ] ILogger_write(
        retval,
        theLog),
        pszLogPass, pszLogFail;

vu_CoUninitialize(); /* Uninitializes COM - maps to vu_CoUninitialize() in sequence diagram */

/* Operations mapped in the sequence diagram */

pop [Think_avg, Think_dist, Think_def, Timeout_val, Timeout_scale];

DATAPool_CONFIG "Pfunctions" OVERRIDE DP_SEQUENTIAL DP_SHARED
    DP_NOWRAP
    {
        EXCLUDE, "iNum", "string", "42";
    }

A.2 Generated Wrapper code

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <comdef.h>
#include <tchar.h>

#ifndef _WIN32_DCOM
#define _WIN32_DCOM
#endif
#define DLLEXPORT __declspec(dllexport)

#import "C:\JLie\Rpa\Test\Datatypes.tlb" no_namespace named_guids

TCHAR sDat1[1024]; /* buffer for strResult */
TCHAR sDat2[1024]; /* buffer for pszLogPass */
TCHAR sDat3[1024]; /* buffer for pszLogFail */
extern "C" {
    DLLEXPORT int intResult;
    DLLEXPORT TCHAR* strResult = sDat1;
void HandleError(_com_error e)
{
    TCHAR buf[1008] = {0};
    _stprintf(buf, "Error: %08x %ls\n", e.Error(), e.ErrorMessage());
    _tcscat(pszLogFail, buf); //com_error message to VU log
    // TODO: Add custom code here.
    // The pszLogFail variable contains data to be automatically logged by LoadTest.
    // You may add your own code here to tailor this data for specific information.
}

DLLEXPORT int IDatatypes_PType_Int(void* Param_int, LONG lplDatatypes = 0)
{
    LONG rc = 1;
    _tcscpy(pszLogPass, _T("Pass: "));
    _tcscpy(pszLogFail, _T("Fail: "));

    INT _Result;
    _Result = 0;
    //Declare and initialize variables for parameters
    //to be passed.
    INT wrParam_int = (INT)_ttol((TCHAR*)Param_int);

    IDatatypesPtr pIDatatypesPtr = NULL;
    IDatatypes* pIObj = (IDatatypes*)lplDatatypes;
    if (pIObj) {
        // Check if the passed-in address is a valid COM interface
        try {
            pIObj->AddRef(); pIObj->Release();
        }
        catch (...) {
            pIObj = 0;
        }
    }

    if (!pIObj) // No valid object was passed in.
        try {
            // Use an existing interface reference if available.
            // To create the object on every iteration, model the
            // vu_CoInitialize and vu_CoUninitialize operations in
// your sequence diagram.

if (pIDatatypes == NULL) {
   pIDatatypesPtr = IDatatypesPtr(__uuidof(Datatypes));

   // Save actual interface ptr in process DLL global memory.
   // Increment the reference count so the smart pointer
   // does not clean up when we lose scope returning control
   // to the script. The vu_CoUninitialize() function
   // releases this reference.

   plDatatypes = pIDatatypesPtr.GetInterfacePtr();
   plDatatypes->AddRef();
}
else {
   pIDatatypesPtr = IDatatypesPtr(plDatatypes);
}

pIObj = plDatatypes;
}
catch (_com_error e) {
   HandleError(e);
   return 0;  // Return 0 to VU to signal a failure.(required)
}
catch (...) {
   _tcscat(pszLogFail, "Failure not COM-related.");
   return 0;
}

try {
   // Call the COM interface using a smart pointer.
   _Result = pIObj->PType_Int(wrParam_int);

   intResult = (int)_Result;
   _stprintf(strResult,"%i",_Result);
   _tcscat(pszLogPass,strResult);
}
catch (_com_error e) {
   rc = 0;  // Return 0 to VU to signal a failure.(required)
   HandleError(e);
}
catch (...) {
   rc = 0;
   _tcscat(pszLogFail, "Failure not COM-related.");
}
return rc;
}

DLLEXPORT int IDatatypes_getLog(LONG lplDatatypes = 0)
{
   LONG rc = 1;
   _tcscpy(pszLogPass,_T("Pass: "));
   _tcscpy(pszLogFail,_T("Fail: "));
ILogger* _Result;
//Declare and initialize variables for parameters
//to be passed.

IDatatypesPtr pIDatatypesPtr = NULL;
IDatatypes* pIObj = (IDatatypes*)lpIDatatypes;
if (pIObj) {
    // Check if the passed-in address is a valid COM interface
    try {
        pIObj->AddRef(); pIObj->Release();
    }
    catch (...) {
        pIObj = 0;
    }
}

if (!pIObj) // No valid object was passed in.
    try {
        // Use an existing interface reference if available.
        // To create the object on every iteration, model the
        // vu.CoInitialize and vu.CoUninitialize operations in
        // your sequence diagram.

        if (pIDatatypes == NULL) {
            pIDatatypesPtr = IDatatypesPtr(__uuidof(Datatypes));
            // Save actual interface ptr in process DLL global memory.
            // Increment the reference count so the smart pointer
            // does not clean up when we lose scope returning control
            // to the script. The vu.CoUninitialize() function
            // releases this reference.
            pIDatatypes = pIDatatypesPtr.GetInterfacePtr();
            pIDatatypes->AddRef();
        } else {
            pIDatatypesPtr = IDatatypesPtr(pIDatatypes);
        }

        pIObj = pIDatatypes;
    }
    catch (_com_error e) {
        HandleError(e);
        return 0; // Return 0 to VU to signal a failure.(required)
    }
    catch (...) {
        _tcscat(pszLogFail, "Failure not COM-related.");
        return 0;
    }
}

try {
    // Call the COM interface using a smart pointer.
    _Result = pIObj->getLog();
}
plLogger = _Result;
intResult = (int)plLogger;
_tcscat(pszLogPass,"Returning the interface pointer as an integer.");
}
catch (_com_error e) {
    rc = 0; // Return 0 to VU to signal a failure.(required)
    HandleError(e);
}
catch (...) {
    rc = 0;
    _tcscat(pszLogFail, "Failure not COM-related.");
}
return rc;
}

DLLEXPORT int ILogger_write(void* bsVal, LONG lpILogger = 0)
{
    LONG rc = 1;
    _tcscpy(pszLogPass, _T("Pass: ");
    _tcscpy(pszLogFail, _T("Fail: ");

    INT _Result;
    _Result = 0;
    // Declare and initialize variables for parameters
    // to be passed.
    BSTR wrbsVal;
    wrbsVal = _bstr_t((TCHAR*)bsVal).copy();

    ILoggerPtr plLoggerPtr = NULL;
    ILogger* pIObj = (ILogger*)lpILogger;
    if (pIObj) {
        // Check if the passed-in address is a valid COM interface
        try {
            pIObj->AddRef(); pIObj->Release();
        }
        catch (...) {
            pIObj = 0;
        }
    }

    if (!pIObj) // No valid object was passed in.
        try {
            // Use an existing interface reference if available.
            // To create the object on every iteration, model the
            // vu_CoInitialize and vu_CoUninitialize operations in
            // your sequence diagram.

            if (pILogger == NULL) {
                plLoggerPtr = ILoggerPtr(_uuidof(Logger));

                // Save actual interface ptr in process DLL global memory.
                // Increment the reference count so the smart pointer

                // Continue with the rest of the function code.
            }
        }
        catch (...) {
            // Handle any errors that occurred during initialization.
        }

    return rc;
}
// does not clean up when we lose scope returning control
// to the script. The vu_CoUninitialize() function
// releases this reference.

pILogger = pILoggerPtr.GetInterfacePtr();
pILogger->AddRef();
}
else {
    pILoggerPtr = ILoggerPtr(pILogger);
}

pIObj = pILogger;
}
catch (_com_error e) {
    HandleError(e);
    SysFreeString(wrbsVal);
    return 0;  // Return 0 to VU to signal a failure.(required)
}
catch (…) {
    _tcscat(pszLogFail, "Failure not COM-related.");
    SysFreeString(wrbsVal);
    return 0;
}

try {
    // Call the COM interface using a smart pointer.
    _Result = pIObj->write(wrbsVal);

    intResult = (int)_Result;
    _stprintf(strResult,"%i",_Result);
    _tcscat(pszLogPass,strResult);
    SysFreeString(wrbsVal);
}
catch (_com_error e) {
    rc = 0;  // Return 0 to VU to signal a failure.(required)
    HandleError(e);
    SysFreeString(wrbsVal);
}
catch (…) {
    rc = 0;
    _tcscat(pszLogFail, "Failure not COM-related.");
    SysFreeString(wrbsVal);
}
return rc;
}

DLLEXPORT int vu_CoInitialize()
{
    ::CoInitialize(NULL);
    return 1;
}

DLLEXPORT int vu_CoUninitialize()
{ 
    // Release the raw interface pointer and set it to 0
    if (pIDatatypes) {
        try {
            pIDatatypes->Release();
            pIDatatypes = 0;
        }
        catch (...) {
            // Catch everything so we do not crash.
        }
    }

    // Release the raw interface pointer and set it to 0
    if (pILogger) {
        try {
            pILogger->Release();
            pILogger = 0;
        }
        catch (...) {
            // Catch everything so we do not crash.
        }
    }

    // Release the raw interface pointer and set it to 0
    if (plLogger) {
        try {
            plLogger->Release();
            plLogger = 0;
        }
        catch (...) {
            // Catch everything so we do not crash.
        }
    }

    ::CoUninitialize();
}
    return 1;
    } }

Bibliography


